An Adaptive Large Neighbourhood Search Algorithm for the Orienteering Problem

Alberto Santini
Universitat Pompeu Fabra and Barcelona GSE, 08005 Barcelona, Spain
alberto.santini@upf.edu

14th September 2018

Abstract

We propose a new heuristic algorithm for the well-known Orienteering Problem, which aims at maximising the prize collected at vertices of a graph, visiting them through a simple closed tour whose length (also known as travel time) is limited by an upper bound. The algorithm is based on the Adaptive Large Neighbourhood Search metaheuristic paradigm, and uses a clustering of the graph to operate on groups of nearby vertices. Extensive computational results show that the proposed algorithm is able to find very good solutions (it produced 27 new best known solutions on a set of benchmark instances) and that it fares favourably compared with other state-of-the-art heuristics when tested with both long and short computation time budgets.

1 Introduction

The Orienteering Problem (OP) was introduced by Golden, Levy and Vohra (1987) and combines the selection of a set of customers to visit with the determination of the shortest tour to visit them. Imagine a tour operator who has to design a walking tour of a city. He has a vast set of possible interest points and associates a score to each of them, based on their desirability. Because the tour time is limited, he has to operate a selection of points to visit and decide the visit order, with the objective of maximising the scores of the visited points.

Besides applications in tourist trip design (Borràs, Moreno & Valls, 2014; Souffriau, Vansteenwegen, Vertommen, Berghe & Oudheusden, 2008; Vansteenwegen & Van Oudheusden, 2007; Wang, Golden & Wasil, 2008), the OP has been considered as a generalisation of the classical Travelling Salesman Problem when the salesman does not have enough time to visit all the cities (Tsiligirides, 1984). In this case, the prizes reflect the expected profit that the salesman can earn in each city. Tsiligirides (1984) also mentions an application in sporting events, which actually gave the name to the OP: an *orienteering event* is a competition usually held in forests where participants start from and come back to a basecamp within a time limit; during their tour they visit intermediate locations where they are awarded points. The participant who returns to the basecamp with the highest number of points wins. The survey by Vansteenwegen, Souffriau and Van Oudheusden (2011) also mentions applications to the home fuel delivery problem (Golden et al., 1987), where customers need to be resupplied with fuel. The lower is the level of fuel at the customer, the higher the urgency of visiting them during the current day; we can then model this problem as an OP where the prizes are inversely proportional to the fuel level. Another application arises in telecommunication network design (Thomadsen & Stidsen, 2003). The ring topology is often used to build wired networks, since it is resistent to single-failure disruptions. The length of the ring is constrained since noise is proportional to the total length and because a ring too long would increase the probability of double-failure disruptions. If the operator assigns a level of importance to each node, for example proportional to the amount of traffic it is expected to route, the problem of linking the most important nodes with a ring network can be modelled as an OP.

A general history of the problem, a description of its many variants, and an overview of exact and heuristic algorithms developed to solve them, can be found in the two surveys by Vansteenwegen et al. (2011) and Gunawan, Lau and Vansteenwegen (2016).

Paper outline. In the rest of this section we formally introduce the problem and summarise the contributions of the heuristic approach we propose. In Section 2 we describe existing algorithms for the OP found in the literature and highlight the neighbourhood structures used. Four of these algorithms, one exact and three heuristics, will be used as a benchmark to evaluate the performance of our algorithm in Section 5. In Section 3

we introduce the problem of clustering the vertices of a graph; such a clustering will be used when describing our algorithm in Section 4. Finally, we draw some conclusions in Section 6 and provide a short list of negative results in Appendix A.

Formal description. To model the problem, consider an undirected graph $G = (V, E)$ where each vertex $i \in V$ has a prize $p_i \in \mathbb{R}^+$, and each edge $\{i, j\} \in E$ has a travel time $t_{ij} \in \mathbb{R}^+$. The objective of the Orienteering Problem is to define a simple closed tour in G maximising the prize collected at each visited vertex, while ensuring that the sum of travel times of the traversed edges is within an upper bound $T \in \mathbb{R}^+$. It is usually assumed that the triangle inequality holds for travel times, that the graph is complete, and that there is a special vertex $0 \in V$, called the *depot*, which needs to be visited. The set of non-depot vertices, also called *customers*, is denoted as $V' = V \setminus \{0\}$.

Contribution. Gunawan et al. (2016) notice how the latest research on the OP has focused on heuristic methods, ranging from swarm optimisation (Sevkli & Sevilgen, 2010) to Large Neighbourhood Search (Liang, Kulturel-Konak & Lo, 2013), from GRASP (Greedy Randomised Adaptive Search Procedure) (Campos, Mart, Sánchez-Oro & Duarte, 2014) to memetic algorithms (Marinakis, Politis, Marinaki & Matsatsinis, 2015). In a recent paper, Kobeaga, Merino and Lozano (2018a) presented a new state-of-the-art genetic algorithm for the OP, which was able to find new best known solutions for large instances. With this work we present a new metaheuristic for the OP based on the Adaptive Large Neighbourhood Search (ALNS) framework and using a clustering of the input graph. We also introduce several new neighbourhoods, including some based on the graph clustering. Computational results show that the method is competitive with state-of-the-art heuristic algorithms: it finds better solutions when given a long time limit, but it also shows very good performances on short 5-minute runs. Furthermore, it seems to be very complementary with the genetic algorithm mentioned above, in the sense that the sets of large instances in which they work best are disjoint. Finally, we were able to produce 27 new best known solutions for the benchmark instances using ALNS and 12 more re-running the genetic algorithm of Kobeaga et al. (2018a) with a different termination criterion.

2 Existing algorithms for the OP

In the paper by Kobeaga et al. (2018a) mentioned in Section 1, the authors implemented a novel genetic algorithm for the OP. They provided an innovative chromosome representation of OP solutions, a new customer inclusion heuristics, and a new crossover method (called edge recombination). The algorithm also involves a local search phase, the "tour improvement operator", which aims at reducing the travel time keeping the set of visited customers fixed. To achieve this goal, the authors propose three possible neighbourhoods: (1) solving heuristically the TSP over the set of vertices included in the tour, using the heuristic of Lin and Kernighan (1973); (2) the 2-opt heuristic (Croes, 1958); (3) the 3-opt heuristic (Lin, 1965). In their customer inclusion heuristic they propose a method to insert a new customer into an existing tour without violating the time bound. Rather than trying all possible positions and choosing the one which increases the travel time less, they reduce the candidate positions set by building a short list of vertices close to the customer to be inserted and only try to place the new customer next to these vertices. This reduces the complexity of exploring this neighbourhood at the expense of risking to miss the optimal move.

The authors made the source code of their algorithm available at (Kobeaga, Merino & Lozano, 2018b); they also expanded the set of benchmark instances first introduced by Fischetti, Salazar-González and Toth (1998) and based on the TSPLib (Reinelt, 1991). This new, larger benchmark set is available at (Kobeaga, Merino & Lozano, 2018c). Furthermore, they obtained the code of three more algorithms for the OP and ran it on a modern machine, to be able to compare their results with those already in the literature, including on the new instances. The three algorithms are:

1. the exact branch-and-cut of Fischetti et al. (1998);
2. the GRASP with Path Relinking of Campos et al. (2014);
3. an iterative algorithm for the Generalised OP (GOP), by Silberholz and Golden (2010).

Notice that this last algorithm is able to solve a more general version of the OP in which each vertex is associated with a prize vector (of dimension, say, v) and the objective is to maximise an arbitrary function $\mathbb{R}^v \to \mathbb{R}$.

We will build on this computational effort and will compare our results to those presented in the mentioned paper. The remainder of this section will give a brief overview of the algorithms for the OP mentioned above, as well as of other algorithms present in recent literature.

The branch-and-cut algorithm of Fischetti et al. (1998) remains the most effective exact algorithm for the OP (Feillet, Dejax & Gendreau, 2005). The authors use a vast array of valid inequalities, separated and added to a base MIP model on the fly, to quickly obtain a very tight LP relaxation and good upper bounds. Since the cut separation procedures can be too time consuming, they use heuristics to separate some of the cuts. Furthermore, they obtain valid lower bounds with an initial heuristic using information from the LP relaxation of the model.

The algorithm of Campos et al. (2014) is based on an initial randomised greedy procedure, which starts from a tour with just the depot and, at each iteration, adds a new customer such that the resulting tour is still feasible. A classic greedy procedure would add the best vertex, where "best" could be defined in terms of various criteria, such as the prize collected or the increase in travel time, or a combination of both. Using the GRASP paradigm, instead, the authors keep a list of good vertices and select one at random from the list. The resulting initial solution is then improved with a local search procedure. This involves the use of two neighbourhoods: one to exchange a vertex in the tour with one outside, without violating the time bound and strictly increasing the collected prize. The other one to reduce the travel time: it searches for vertices not in the tour which have the same prize of some vertex in the tour, and exchanges them if this results in a shorter tour. Because the initial solution depends on random choices, the whole algorithm is repeated multiple times and the final solutions are collected together with indermediate ones. At the end, the authors apply a path-relinking procedure (Resende & Ribeiro, 2005) to each pair of stored solutions and return the best resulting solution.

The iterative algorithm of Silberholz and Golden (2010) has features resembling the *ruin-and-recreate* metaheuristic (Schrimpf, Schneider, Stamm-Wilbrandt & Dueck, 2000). The authors first build a tour using a simple greedy procedure. Then, they iteratively destroy it by removing some customers from the tour and repair it by inserting other customers (the removed customers can potentially be re-inserted, but they have the lowest priority). This ruin and recreate cycle is then repeated until the solution has not improved for a fixed number of iterations. Their algorithm depends on two parameters: the number of vertices to remove during the ruin phase and the number of iterations without improvement used as a termination criterion.

Sevkli and Sevilgen (2010) present a Particle Swarm Optimisation (PSO) method for the OP. To cope with the discrete nature of the problem, they redefine the classical PSO operators and thus obtain a Discrete PSO heuristic. They strengthen the algorithm with a local search step based on the Reduced Variable Neighbourhood Search paradigm (Hansen & Mladenovi, 2003). They use three neighbourhood structures, in the following order: increasing profit, decreasing travel time, and 2-opt. The first structure choses a random position in the tour and tries to increase the collected profit by adding one new customer and, eventually, removing an old one. The second structure is analogous to the first one, but aims at reducing the tour length. The third structure implements the classic 2-opt heuristic.

Liang et al. (2013) used a Variable Neighbourhood Search approach to define a new heuristic algorithm for the OP. Similar to the approach of Sevkli and Sevilgen (2010), they use two types of neighbourhoods: one to reduce the tour length, and one to increase the collected prize. They define three neighbourhoods of the first type: (1) swapping two vertices within the tour, (2) 2-opt, and (3) moving a vertex within the tour. Additionally, there are two neighbourhoods of the second type: (1) replace a vertex in the tour with one outside, and (2) insert a non-visited vertex in the tour. After the local search phase with these neighbourhoods, they perturb the solution removing random vertices from the tour.

A hybrid between GRASP and a memetic algorithm (a type of evolutionary algorithm) was proposed by Marinakis et al. (2015). The GRASP procedure is used to generate the initial population, while the memetic algorithm tries to combine the individuals to produce offspring of better quality, i.e., tours collecting a higher prize. It is interesting to notice that the initial population generation includes a local search phase using two neighbourhoods, one to rearrange the nodes of the tour to decrease its travel time, and one to insert new customers into the tour. The authors, however, do not give additional details on these neighbourhoods.

Finally, we mention that this is the first time, to the best of our knowledge, that an ALNS-based algorithm is used to solve the OP. There are, however, recent application of the ALNS paradigm to similar problems. One is the pickup-and-delivery problem with time windows, profits (hence the similarity), and reserved requests (Li, Chen & Prins, 2016). Another on is the service technician routing problem (Kovacs, Parragh, Doerner & Hartl, 2012), which resembles a Team Orienteering Problem. This is a variant of the OP with multiple vehicles, in this case corresponding to multiple technicians. Because some customers can be outsourced at a cost, the two problems are similar: we can alternatively see the visited customers as earning a profit equal to the savings deriving from not outsourcing the service. We also mention an inventory-routing problem arising at an organisation producing biodiesel from oil (Aksen, Kaya, Salman & Tüncel, 2014). The decision-maker needs to decide which facilities to visit to collect exhaust oil. If not enough exhaust oil is collected, the fuel plant will need to purchase more expensive virgin oil. Therefore, there is a tradeoff between the collection costs, which play the role of the travel time in our description of the OP, and the prize collected at nodes, which corresponds to the difference in price between the virgin and exhaust oil.

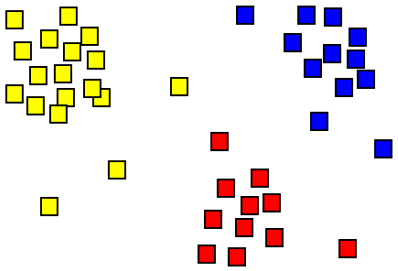


Figure 1: Spatial clustering example. Image source: Wikimedia Commons. Public Domain.

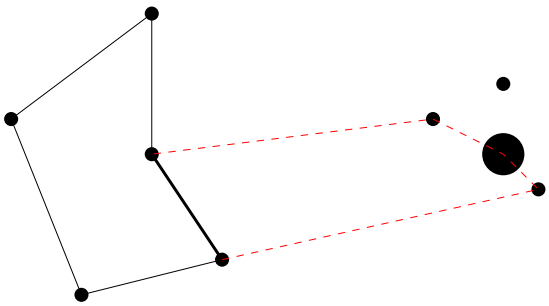


Figure 2: Including a promising vertex in a tour, together with some of its neighbours.

3 Clustering and the OP

Clustering is a machine learning task whose objective is to assign labels to objects, so that objects with the same label are closer to each other than to objects with other labels. A group of objects with the same label is called a *cluster*. The precise meaning of "closer" in the definition above is problem-specific, and can be formalised by assigning a distance function to the set of objects. Figure 1 shows an example of spatial clustering, in which the distance function is the euclidean distance. Clustering the objects in three clusters (red, yellow, blue) results in visually compact sets of points. A clustering such as that depicted in the figure, where all objects are labelled, is called *complete*. On the other hand, an incomplete clustering can have some objects with no label, which are called *outliers*.

The relationship between clustering and the OP are not immediately clear, and deserves some explanation. The most fascinating characteristic of the OP is probably the tension between two conflicting metrics: the maximisation of collected prize, and the reduction of travel time. Given a feasible tour with sufficient slack on the travel time bound, we might ask which customers to include in the tour to maximise the collected prize. Imagine that visiting a customer $i \in V'$ which lies far away from the tour is both desirable and feasible (for example, it has been chosen via a savings heuristic such as the classic Clarke and Wright (1964) method). Then we would try to include in the tour other customers close to i, since we have already made a considerable travel-time investment to reach i's area. Figure 2 gives a visual representation of this idea. In the figure, vertices are represented with dots and their prizes are proportional to the dots' areas. Travel times are proportional to the euclidean distance between dots. Imagine we have the starting tour drawn in black on the left of the figure, and we have enough time to spare to visit the big black dot (customer i) on the right. With a minimal additional time investment, we can also visit some of the neighbours of i. In the figure we visit two additional customers, which are very close to i. The amended tour will include the red dashed edges, while removing the thick black edge. Generalising, we can say that customers that can be clustered together have a high probability of being included in the tour if one of them is included. We will use this property in Sections 4.4 and 4.5 to decide how to remove/add customers from/to an existing tour.

Now, however, a new question arises: how to cluster the customers of G? Many popular clustering algorithms (such as the popular k-means algorithm (Lloyd, 1982)) require the number of desired clusters as an *input* parameter. This requirement is not compatible with the purpose of using clustering to group together nearby customers, since the number of desired clusters is not known a priori, and could differ dramatically from instance to instance. For this reason, we decided to use the *Dbscan* algorithm of Ester, Kriegel, Sander and Xu (1996). This is a popular algorithm in the machine learning community, which does not need the number of clusters as a parameter. It produces a possibly *incomplete* clustering of points, given two input parameters: a radius $r \in \mathbb{R}^+$, and a number $N \in \mathbb{N}^+$. A point $j \in V'$ is considered a neighbour of a point $i \in V'$ if their distance is not larger than the radius. Points which have at least N neighbours (including themselves) are called *core points*. Every neighbour of a core point is called a *reachable point*. Notice that a reachable point can be a core point itself, if it also has at least N neighbours. Points which are neither core nor reachable are *outliers*. Clusters are formed starting from one core point and recursively adding all points reachable from any core point already in the cluster.

Figure 3 gives an example of clustering with $N = 3$ and r equal to the radius of the dashed circles. Segments between points indicate a neighbouring relationship. The three red points are core points, since each of them has three neighbours (including themselves). Blue points are reachable from core points, but are not core points themselves, since they only have two neighbours each. Finally, the black points are outliers. A cluster is then

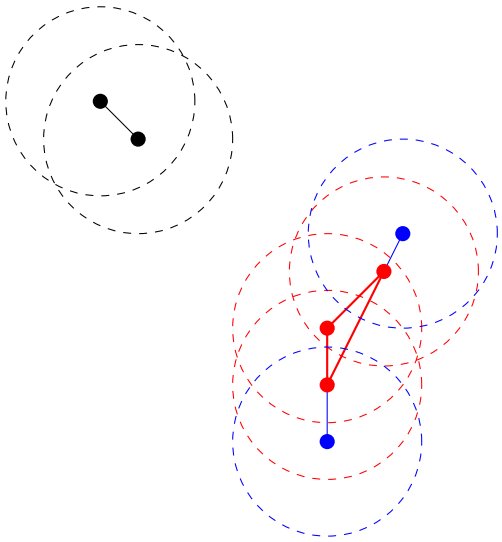
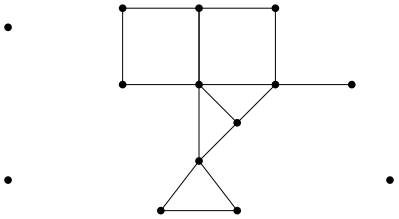
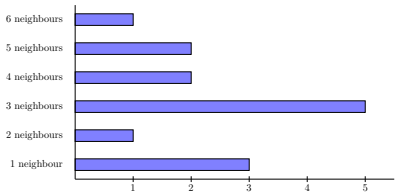


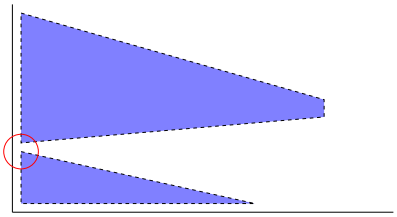
Figure 3: Dbscan example with one cluster and two outliers. Red points are core, blue ones are reachable, black ones are outliers.



(a) Spatial distribution of the vertices. A line between two vertices indicates they are neighbours.



(b) Histogram showing the distribution of the number of neighbours for the vertices.



(c) General shape of the histograms.

Figure 4: Example of neighbourhoods over 14 vertices.

a subgraph induced by the neighbouring relationship, containing at least one core point. In the example of Figure 3 we would only have one cluster, the one in the bottom-right part of the figure.

While Dbscan also depends on input parameters, namely r and N, we argue that it is easier to estimate automatically these parameters, compared with the "number of clusters" parameter. We used the following procedure to automatically derive suitable values. We first consider the distance from each customer to its nearest neighbour (excluding itself):

$$d_i = \min_{j \in V', j \neq i} d(i, j)$$

where $d(i, j)$ represents the distance between vertices i and j. We then set r as the smallest such distance, $r = \min\{d_i,\ i \in V\}$. In this way, each vertex has at least one neighbour and, therefore, a chance to belong to some cluster. To determine N, we first compute how many neighbours each vertex has, given the choice of r just made. To this end, let:

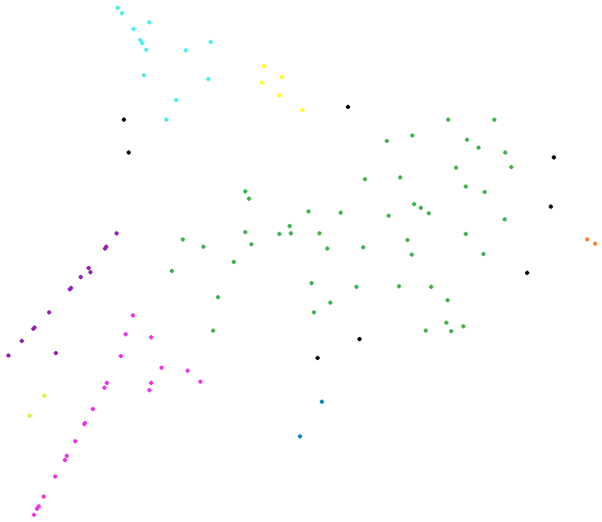
$$N_i = |\{j \in V' \ :\ d(i, j) \leq r\}|$$

We then sort the values N_i and count how many times each of them occurs.

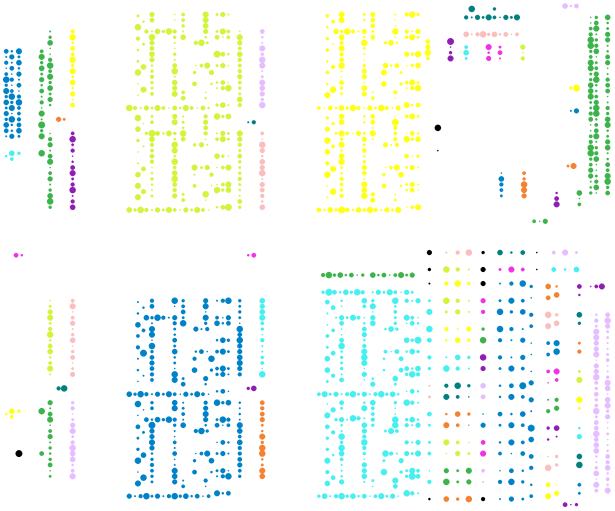
For example, given the vertices in Figure 4a, where a line is drawn between two edges if they are neighbours, we would have the following values for N_i:

$$6\ 5\ 5\ 4\ 4\ 3\ 3\ 3\ 3\ 3\ 2\ 1\ 1\ 1$$

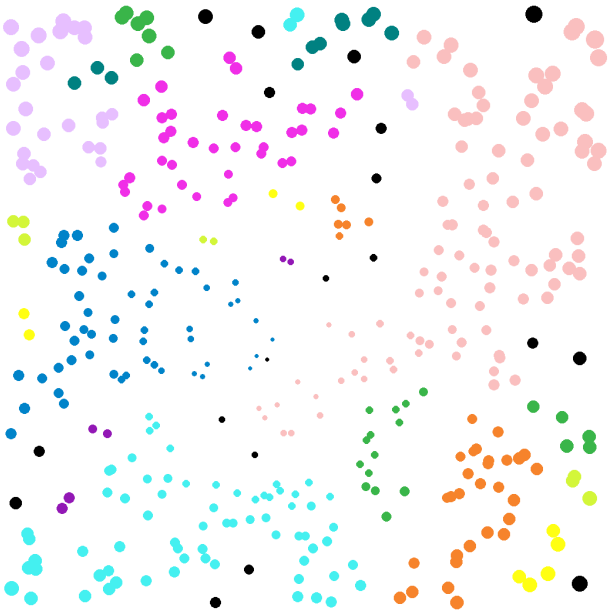
Figure 4b shows a histogram for these value. To limit the number of bars in the histogram, in case of large graphs, instead of considering all possible values taken by N_i, we bucket them in at most twenty buckets uniformly dividing the range between the lowest and the highest N_i. Somehow surprisingly, plotting such histograms reveals that most of the time, while the values on the x and y-axes are different, the shape of the histogram itself is the same. In particular, there are often a large number of points in the lower buckets (few neighbours), while the rest of the histogram resembles a normal or a triangular distribution, with a peak at some intermediate value (see Figure 4c). In the example of Figure 4b, this value would be "3 neighbours". On the basis of this observation, we formulate the following rule of thumb to determine the value of N. Starting



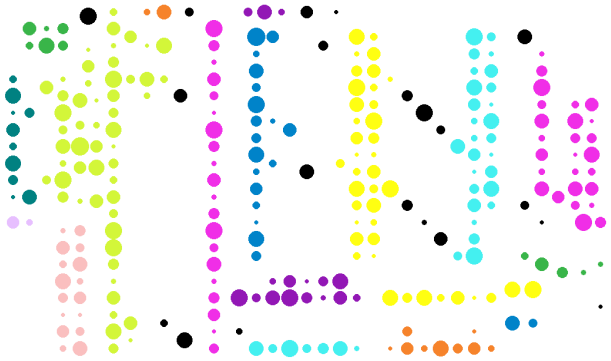
(a) Instance `gr120-gen1`. Depot not shown.



(b) Instance `d1655-gen2`. Depot not shown.



(c) Instance `rd400-gen3`.



(d) Instance `a280-gen4`.

Figure 5: Clustering of OPLib instances.

from the lowest bucket, take the first bucket such that the bucket above it has a strictly larger size, eventually skipping any empty bucket. This rule has been determined empirically and is very simple, yet it allows to obtain very good results in terms of the final clustering of the graph. (In other words, it is able to produce clusters which are visually compact and often contain lined-up points, which could then be visited with a minimal time investement — see, e.g., the pink cluster on the left of Figure 5d.) In the example of Figure 4, the selected bucket would then be the one corresponding to "2 neighbours", and we would set $N = 2$. When a bucket represents a range of values, we take the upper limit of this range.

Examples of clustering OPLib (Kobeaga et al., 2018c) instances using this method are presented in Figure 5. Notice that the colour palette consists of 11 colours, so different clusters could have the same colour in the figure, if more than 11 clusters were produced. The size of the dots is directly proportional to their prize. Black dots indicate outliers, and we first removed the depot from the graph.

Algorithm 1: ALNS Metaheuristic

Input : Number of iterations: M
Input : Initial solution: x_0
Input : List of destroy methods: $\mathcal{D}$
Input : List of repair methods: $\mathcal{R}$
Input : Destroy method scores: $\lambda_D,\ \forall D \in \mathcal{D}$
Input : Repair method scores: $\lambda_R,\ \forall R \in \mathcal{R}$
Input : Local Search method: L
Input : Objective in maximisation form: $f(\cdot)$

1 $x = x_0$
2 $x^* = x_0$
3 $i = 1$
4 $\lambda_D = 1,\ \forall D \in \mathcal{D}$
5 $\lambda_R = 1,\ \forall R \in \mathcal{R}$
6 **while** $i \leq M$ **do**
7 Choose $D \in \mathcal{D}$ with probability proportional to λ_D
8 Choose $R \in \mathcal{R}$ with probability proportional to λ_R
9 Select $x' = R(D(x))$
10 Optional: select $x' = L(x')$
11 **if** *Accept new solution* x' **then**
12 $x = x'$
13 **end**
14 **if** $f(x) > f(x^*)$ **then**
15 $x^* = x$
16 **end**
17 Update scores λ_D, λ_R
18 $i = i + 1$
19 **end**
20 **return** x^*

4 An ALNS-based heuristic for the OP

In this section we introduce a heuristic for the OP, based on the popular Adaptive Large Neighbourhood Search (ALNS) paradigm of Ropke and Pisinger (2006). We start by briefly explaining the structure of an ALNS-based heuristic; we will then introduce the specific operators devised for the OP.

The basic idea behind ALNS is that, given a generic feasible space X, we can move from a solution $x \in X$ to another one, $x' \in X$, by first destroying part of x, and then repairing it. The intermediate destroyed solution need not be a member of X; in other words, it can be infeasible. In general, there are many sensible ways to both destroy a feasible solution and repair a destroyed one. Since the effectiveness of these methods can be different for each instance of the problem, ALNS proposes to keep a battery of *destroy* and *repair methods*, and learn during the solution process which ones give the best results. This is done by associating each method to a score. Scores are all equal at the beginning, but are then increased for methods producing good solutions. At each iteration, the methods are chosen randomly, with a probability proportional to their score. Notice that this procedure can be seen as a crude learning algorithm, where we try to learn the optimal probabilities associated to each method.

A formal description of this metaheuristic is given in Algorithm 1, where the aim is the maximisation of a function f. The current solution and the best known solution are initially set to the start solution x_0, respectively in Line 1 and Line 2. Line 3 initialises the iteration count, while Lines 4 and 5 give all detroy and repair methods the same starting score. At each iteration, a destroy and a repair method are chosen at Lines 7 and 8, and a new solution is produced by applying them (Line 9). Optionally, the new solution can be improved using a local search method (Line 10). Because local search is usually expensive, this step could be performed only at set times; for example, only when a new global best is found. We then have to determine whether the new solution should replace the current one or be discarded, according to some *acceptance criterion* (see Section 4.1), as shown in Lines 11 and 12. Furthermore, if the new solution improves on the best solution found so far, we store its value (Line 15). In Line 17 we update the destroy and repair methods' score, as explained in Section 4.2, and in Line 18 we increase the iteration counter. Finally, when the loop has run for M iterations, we return the best encountered solution (Line 20). Notice that reaching the total number of iterations is not the only possible termination criterion. Other criteria are, for example, reaching a maximum computation time (i.e., a *timeout*) or running for a prefixed number of iterations without improving x^*.

4.1 Acceptance criterion

The acceptance criterion is a crucial part of the ALNS algorithm. Santini, Ropke and Hvattum (2018) have shown that choosing one acceptance criterion over another (among the many that have been proposed in the literature) has statistically significant effects on the overall algorithm performance. The authors also made some recommendations on the choice of the acceptance criterion. In particular, they showed that simple and easy-to-tune methods such as linear record-to-record travel *Lin-RRT* are not worse than classical criteria based on simulated annealing. For this reason, we decided to use the Lin-RRT acceptance criterion, based on the record-to-record travel heuristic of Dueck (1993).

This criterion demands that the new solution x' be accepted if it is either better than x^*, or worse than x^* but within a gap of $T \in [0, 1]$. In other words, x' is accepted if

$$\frac{f(x^*) - f(x')}{f(x^*)} < T \tag{1}$$

The threshold T varies during the solution process, from a starting value T^{s} to an ending value T^{e}. In the linear variant, the threshold moves linearly from T^{s} to T^{e} based on the current iteration number. The advantage of this choice compared to, say, an exponential decrease, is that we can simply fix $T^{\mathrm{e}} = 0$ resulting in just one parameter to tune. This choice is justified empirically in (Santini et al., 2018) noticing that when performing parameter tuning on both parameters, T^{e} usually ends up at or close to 0.

4.2 Scores update

The update of the destroy and repair scores happens in the following way. Let $F \in \mathcal{D} \cup \mathcal{R}$ be any destroy or repair method. At each iteration, if F is used, its score is updated according to the formula

$$\lambda_F = h\lambda_F + (1 - h)\varphi \tag{2}$$

where $h \in [0, 1]$ is a *decay* parameter, controlling how fast the score can change from one iteration to the next, and $\varphi \in \mathbb{R}^+$ is a score associated to the performance of the method during the last iteration. It can assume three values:

$$\varphi = \begin{cases} \omega_1 & \text{if using } F \text{ produced a new global best solution} \\ \omega_2 & \text{otherwise, if } F \text{ produced a solution better than the current one} \\ \omega_3 & \text{otherwise, if } F \text{ produced a solution which was accepted} \end{cases}$$

such that $0 \geq \omega_1 \geq \omega_2 \geq \omega_3$. The score is not updated if the new solution was not accepted.

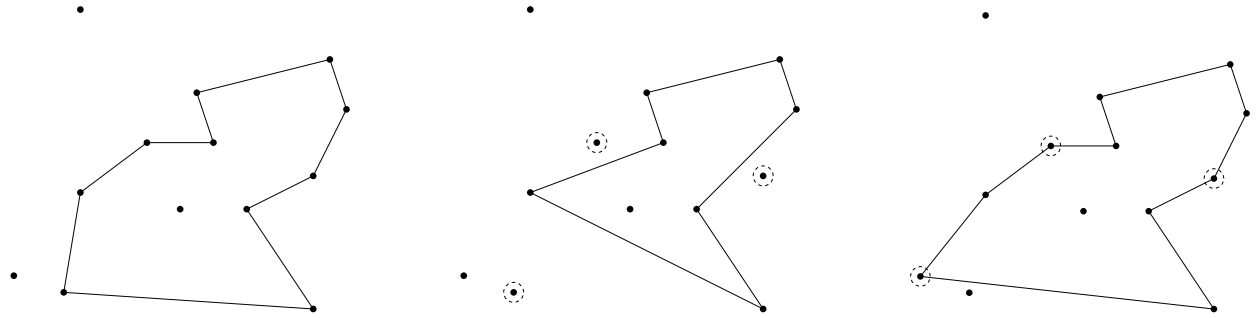
4.3 Initial solution generation

An initial solution is produced with a simple randomised constructive procedure. A starting solution is defined by only including the depot in the tour. We then shuffle the customer set V' and try to insert the customers one by one, in the given order. Each vertex gets inserted in the tour at the position which results in the smallest increase in travel time. If a vertex cannot be inserted at any position without violating travel-time feasibility, it is skipped.

4.4 Destroy methods

As mentioned before, a destroy method receives a feasible solution and destroys part of it, returning a potentially infeasible one. In our case, since we only remove customers from a feasible tour, we are always returning another feasible solution. We devised three destroy methods, described in the following.

Random remove. Given a tour $(0, i_1, \ldots, i_k, 0)$ this method simply removes $\alpha^{\mathrm{rr}} \cdot k$ customers selected uniformly at random. The number $\alpha^{\mathrm{rr}} \in (0, 1)$ is a parameter, which defines how "aggressive" the method is in destroying the solution.

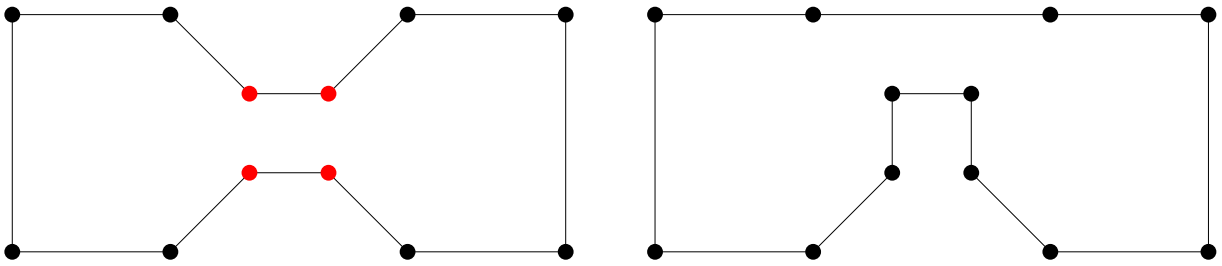


(a) Initial tour before random remove is applied.

(b) Tour destroyed by random remove.

(c) Possible repaired tour.

Figure 6: An example of when the random remove method might fail to introduce enough diversity. Dashed circles indicate the removed or inserted vertices.



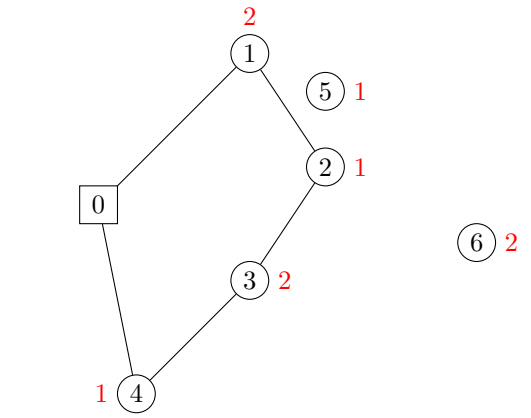
(a) Initial tour before cluster remove is applied.

(b) A possible repaired tour.

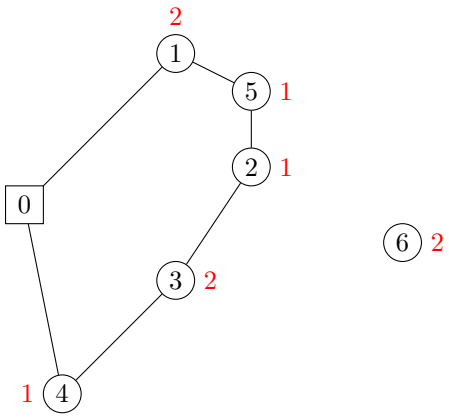
Figure 7: An example of when the Random cluster remove method might be more effective than Random sequence. Red dots indicate a cluster of vertices removed from the initial tour.

Random sequence remove. One potential disadvantage of the random remove method is that sometimes there might not be much margin during the repair phase to produce a solution different from the starting one. Figure 6 gives an example of this behaviour. The figure on the left shows the tour before the destroy method is applied. In the central figure, the tour has been destroyed and three vertices have been removed. The rightmost figure shows a possible repaired tour. Notice that the initial and the final tour only differ by one vertex, as for the other two vertices the most sensible choice (in a "greedy" sense) was to simply reinsert them in the same positions. This problem is correlated with two characteristics of the removal process. The first is that not enough vertices are removed. If we were to increase the number of removed vertices too much, however, the destroy-repair move would start to look increasingly as a cold-started greedy heuristic, so there is only a small margin for us to increase α^{rr} without dramatically decreasing the quality of the algorithm. The second is that the removed vertices are distributed uniformly around the tour. In fact, if we wanted to reinsert a removed vertex and we did not perform any other removal of nearby vertices, most of the time the greedy optimal choice would be to just place it back where it was. We can mitigate this second factor by deciding that, once the number of vertices to remove is fixed, the chosen customers appear consecutively in the tour (eventually interleaved by the depot). To this end, we randomly choose a customer i in the tour and remove $\alpha^{\text{rr}} \cdot k$ customers starting from i. This destroy method is called *random sequence remove*.

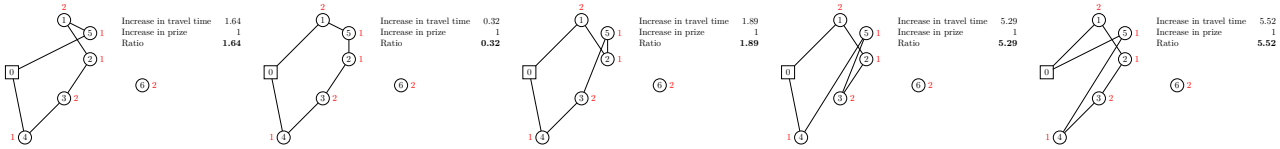
Random cluster remove. The last destroy method is similar to random sequence remove but, instead of removing vertices which appear consecutively in the tour, it removes vertices belonging to a same cluster. This approach, too, increases the chances that the removed vertices appear at nearby positions in the tour, but it also allows for other ways to destroy (and then repair) the tour, compared to the previous method. An example is given in Figure 7, where the red vertices form a cluster. Selecting that cluster, we remove vertices which are nearby but appear at distant positions in the original tour. In general, let $C_1, \ldots, C_m \subset V'$ represent a (possibly incomplete) clustering. A random cluster C_l is selected uniformly over all clusters. If the size of the intersection between the cluster and the current tour $(0, i_1, \ldots, i_k, 0)$ is not too big, i.e. if $|C_l \cap \{i_1, \ldots, i_k\}| \leq \alpha^{\text{rr}} \cdot k$, all customers in said intersection are removed from the tour. Otherwise, $\alpha^{\text{rr}} \cdot k$ customers are randomly sampled from the intersection and are then removed.



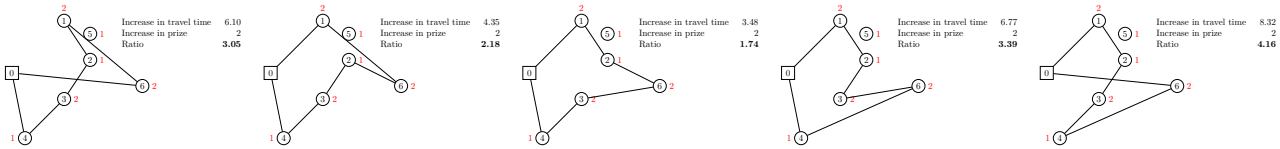
(a) An initial tour and two possible customers to insert. Red numbers indicate prizes.



(b) The tour at the end of the first iteration of Greedy repair, after applying the chosen insertion.



(c) Possible insertions of customer 5 and relative calculations of the ratio between increased travel time and increased profit.



(d) Possible insertions of customer 6 and relative calculations of the ratio between increased travel time and increased profit.

Figure 8: An example of insertion of new customers into a tour with the Greedy repair method. At the current iteration we would insert customer 5 between customers 1 and 2, as this insertion corresponds with the lower ratio (0.32).

4.5 Repair methods

The repair methods try to increase the profits collected, bringing in more customers. In some cases, during an intermediate step, this is done by accepting tours which are time-infeasible, and feasibility is then restored before returning the new tour. We devised four repair methods, described in the following.

Greedy repair. The greedy repair method considers all vertices which are not part of the tour, and builds a list of all possible positions where each vertex could be inserted, provided that they do not violate the time-feasibility of the solution. All these positions are scored considering the ratio between the increase in travel time and the increase in collected prize if we were to perform an insertion at said position. We define an *insertion* as a pair (i, k): i is the customer to add, and k the position of the vertex of the tour after which we add i. The insertion with the lowest ratio is chosen, and insertions either (a) relative to the same customer i, or (b) relative the same position k, or (c) which became time-infeasible, are removed from the list. Finally, the scores of insertions in positions adjacent to the chosen one (k and $k + 1$) are recomputed. The procedure is repeated until the list is empty.

Figure 8 shows how the moves are evaluated when applying the Greedy repair method. Starting from the initial tour in Figure 8a with two unvisited customers, in the first iteration of the method we aim at inserting one of them in a favourable position. We compute the ratio between increases in travel time and prize, associated with all possible insertions of customer 5 (Figure 8c) and 6 (Figure 8d). The one with the lowest ratio corresponds to inserting customer 5 between customers 1 and 2, producing the new tour depicted in Figure 8b.

Random repair. With the random repair method, we first list all customers that are not part of the tour. We then draw a random number in $[0, 1]$ and use it as the fraction of vertices in the list which will actually be inserted in the tour. For example, if the random number is 0.25, we will only insert 25% of the vertices, chosen randomly. Each vertex is inserted, even if doing so makes the tour time-infeasible. The vertex, however, is always inserted in the position which increases the total travel time the least. At the end of this procedure, if the resulting tour actually violates the constraint on total time, feasibility is restored as described in Section 4.8.

Prize repair. The prize repair method is very similar to random repair, except that — once the fraction of vertices to insert has been chosen — the corresponding vertices are not chosen randomly, but by prize. As in the example above, if 0.25 is drawn, then we will insert in the tour the 25% of the vertices with the highest prize.

Cluster repair. This repair method aims at inserting in the tour vertices which are close to each other and therefore can add a good prize with a small time investment. We first choose a random cluster and consider all vertices of the cluster which are not already in the tour, in random order. All these vertices are inserted (even if doing so makes the tour time-infeasible), and each vertex is placed in the position which increases the travel time the least. At the end, feasibility is eventually restored with the method described in Section 4.8.

4.6 Neighbourhoods

The repair methods described above are all compatible with all the destroy methods, in the sense that any destroyed solution can be repaired by any repair method. Because of this, the 3 destroy and 4 repair methods implicitly define $3 \cdot 4 = 12$ explicit neighbourhoods. Some of the methods we use bear resemblance with neighbourhoods already present in the literature: Random destroy to the perturbation phase of Liang et al. (2013); Greedy repair and, to a lesser extent, Random repair and Prize repair share features with methods used by Campos et al. (2014), Marinakis et al. (2015), Sevkli and Sevilgen (2010), Silberholz and Golden (2010). The other methods — namely Random sequence remove, Random cluster remove, and Cluster repair — are, to the best of our knowledge, being introduced for the first time.

4.7 Local search

We devised three possible local search steps to apply whenever the algorithm produces a new promising solution. Since local search can be time consuming, we decided to only apply it when the best solution (x^* in Algorithm 1) is updated, which happens just a few times during the solution process.

The first possibility is to simply try to insert as many more vertices as possible into the tour if there is some time slack (we call this procedure "Fill"). In practice, this is accomplished by applying the greedy repair method to the new solution, in case the solution was produced by a different repair method. The second possibility is to first apply the famous 2-opt heuristic (Croes, 1958) to the tour, and then use Fill. We call this procedure "2OptFill". Finally, the third possibility is to solve an instance of the Travelling Salesman Problem over the selected vertices, and then use Fill. We call this procedure "TSPFill". In our case, the TSP is not solved optimally, but applying the well-known Lin-Kernighan heuristic (Lin & Kernighan, 1973), with the improvements proposed by Helsgaun (2000) which involve "larger and more complex search steps and the use of sensitivity analysis to direct and restrict the search".

4.8 Restoring the travel-time feasibility of a tour

Given a tour which is not travel-time feasible, we can restore feasibility removing vertices from the tour. Let the tour be $(0, i_1, \dots, i_{q-1}, i_q, i_{q+1}, \dots, i_k, 0)$. We associate to each vertex i_q a score given by the ratio between the decreases in travel time and in price that we would obtain removing i_q from the tour. In other words, the score is

$$\frac{t_{i_{q-1}i_q} + t_{i_q i_{q+1}} - t_{i_{q-1}i_{q+1}}}{p_{i_q}}$$

The vertex $i_{\bar{q}}$ with the highest score is removed, scores are re-computed (notice that this is only necessary for $i_{\bar{q}-1}$ and $i_{\bar{q}+1}$), and the procedure is repeated until the tour becomes feasible.

5 Results

The results presented in this section have been obtained on a cluster with quad-core Intel Xeon E5 processors, running at 2.20GHz. We booked one cpu and 2GB of RAM for each run, except for instances `pla7397-gen2` and `pla7397-gen3` for which we booked 4GB of RAM (the programme was otherwise running out of memory). The algorithm was coded in `C++`. The source code is available on GitHub (Santini, 2018).

Algorithm	Reference	Type	Runs	Termination
B&C	Fischetti, Salazar-González and Toth (1998)	Exact	1	Time (5 hours).
GRASP	Campos, Mart, Sánchez-Oro and Duarte (2014)	Heuristic	10	Either 500 construction phases + relinking, or time (5 hours); whichever kicks in first.
Iterative	Silberholz and Golden (2010)	Heuristic	10	4500 iterations without improvement.
Genetic	Kobeaga, Merino and Lozano (2018a)	Heuristic	10	Either time (5 hours) or fitness (25% of population has the same fitness as the best solution); whichever kicks in first.
ALNS	This paper	Heuristic	10	Either time (5 hours) or iterations without improvement (250 000); whichever kicks in first.

Table 1: Summary of the algorithms compared in this section, including the type of the algorithm (exact or heuristic), the number of re-runs considered when reporting computational results, and the the termination criterion used.

5.1 Test instances

Kobeaga et al. (2018a) have proposed a new benchmark library for the Orienteering Problem, called OPLib (Kobeaga et al., 2018c). They started from an existing set of instances (Fischetti et al., 1998), and extended it in two directions: 1) they created new instances selecting those generator settings resulting in samples which are hard to solve with exact methods, basing this evaluation on the branch-and-cut algorithm of Fischetti et al. (1998); 2) they added brand-new instances based on graphs larger than those already used in the starting set.

The OPLib contains four sets of instances, called "generations", each with 86 instances of either medium (45 instances, $\leq$ 400 vertices) or large size (41 instances, $>$ 400 vertices). While the base graph topology is the same for instances belonging to the various generations, the method used to assign prizes to the vertices and to determine the travel time bound differ:

- In generation 1, all vertices have unit prize.
- In generations 2 and 4, prizes are calculated according to the formula: $p_i = 1 + (7141(i-1) + 73) | 100$, where $|$ denotes the modulo operator.
- In generation 3, the formula is: $p_i = 1 + \left\lfloor \frac{99 \cdot d(0,i)}{\max_{j \in V'} d(0,j)} \right\rfloor$.

In generations 1, 2, and 3, the travel time bound is set as $T = \frac{1}{2} \cdot \text{TSP}^*(V)$, where TSP^* denotes the travel time of the optimal TSP solution over the whole graph. The constant $\frac{1}{2}$ was chosen in (Fischetti et al., 1998), as it was noticed that OP instances for which the optimal solution visits roughly half the total number of vertices tend to be harder. In generation 4, Kobeaga et al. (2018a) actually tried different values for this constant, and chose the one giving the *hardest* instance. The hardness of the instance was measured solving it with the branch-and-cut algorithm of Fischetti et al. (1998).

5.2 Comparison with existing algorithms

In this section, we compare the results obtained with our algorithm to those obtained by Kobeaga et al. (2018a) for their genetic algorithm, as well as for the the other three algorithms described in Section 2. In that paper, in fact, the authors ran the four algorithms on a modern computing environment, very similar to our own (Intel Xeon E5 at 1.9GHz and 4GB of RAM).

Table 1 gives an overview of the five algorithms compared in this section. Information for "B&C" (branch-and-cut), "GRASP", "Iterative" and "Genetic" are derived from (Kobeaga et al., 2018a), where the exact algorithm was run by the authors only once while, for the heuristics, they report results relative to the best of 10 runs. Notice that the Genetic Algorithm is run for 5 hours, or until the first quartile of the population has the same fitness as the best individual. This termination criterion aims at stopping the algorithm early, if there is not enough diversity in the population and, therefore, the best solution value reaches a large plateu. This is the case, for example, if the best solution is globally optimal, or if it is locally optimal but in a deep "valley". To have a termination criterion similar in spirit to the one used by Kobeaga et al. (2018a) we use, on top of a time limit, a maximum number of iterations without improvement to the best solution.

The overall termination criterion for ALNS is not independent on the instance size: larger instances require more time per iteration, so they will tend to use the full time limit, whereas execution on smaller instances will likely terminate because of the iteration limit. Furthermore, large neighbourhood methods can escape local

Parameter	Description	Type	Reference	Tuning range	Tuned value
T^{s}	Start threshold	ALNS	Section 4.1	$[0, 1]$	0.0039
h	Score decay	ALNS	Section 4.2	$[0, 1]$	0.4314
$\omega_1, \omega_2, \omega_3$	Scores	ALNS	Section 4.2	$\mathbb{R}^+$	$3.0383, 5.3385, 15.3815$
α^{rr}	Remove fraction	Problem	Section 4.4	$[0, 1]$	0.2062
Local Search	Local search	Problem	Section 4.7	Fill, 2OptFill, TSPFill	Fill

Table 2: Algorithm parameters, their tuning range, and their tuned values. Long timeout (30 minutes).

Parameter	Description	Type	Reference	Tuning range	Tuned value
T^{s}	Start threshold	ALNS	Section 4.1	$[0, 1]$	0.0138
h	Score decay	ALNS	Section 4.2	$[0, 1]$	0.2032
$\omega_1, \omega_2, \omega_3$	Scores	ALNS	Section 4.2	$\mathbb{R}^+$	$1.9895, 3.4503, 38.7395$
α^{rr}	Remove fraction	Problem	Section 4.4	$[0, 1]$	0.0643
Local Search	Local search	Problem	Section 4.7	Fill, 2OptFill, TSPFill	TSPFill

Table 3: Algorithm parameters, their tuning range, and their tuned values. Short timeout (5 minutes).

optimality "valleys" more easily than evolutionary algorithms, so the best solution is expected to keep improving for longer, and therefore the chosen criterion is expected to penalises ALNS in terms of runtime, but possibly produce better solutions in the end. For these reasons, after running a first set of computational experiments with these termination criteria, which are useful to compare ALNS with the other four algorithms studied in (Kobeaga et al., 2018a), we ran a second set of experiments, with a more uniform criterion: a 5-minute timeout, without any other condition.

5.2.1 Parameter tuning

The algorithm presents a number of parameters that need tuning. These can be divided in problem-independent parameters, which we "inherit" from using the ALNS framework, and problem-dependent parameters, which are the ones directly related to our problem. To perform the tuning, we used the *irRace* software by López-Ibáñez, Dubois-Lacoste, Pérez Cáceres, Stützle and Birattari (2016). We used 16 tuning instances (four randomly chosen instances from each of the four generations) and gave *irRace* a tuning budget of 5000 experiments. The tuning budget corresponds to the maximum number of times the solver programme can be called (iRace User Guide, 2018, Sec. 10.1). Since we consider two very different set of experiments, we performed tuning twice: once for the longer time limit, and once for the shorter one. Tuning with a 5-hour time limit, however, proved computationally too time-consuming, given the large number of experiments that have to be performed. For this reason, during tuning we reduced the longer time limit from 5 hours to 30 minutes.

The results of the two tuning procedures are summarised in Tables 2 and 3. For the long time limit, the acceptance threshold T^{s} is more stringent and less emphasis is put on the method which finds a new best solution (parameter ω_3). For the short time limit, more emphasis is put on speed: by reducing the number of vertices to remove during the destroy phase (parameter α^{rr}) the repair methods will operate faster. The corresponding savings in runtime are then spent on a more aggressive local search phase, using the TSPFill method.

5.2.2 Effect of destroy and repair methods

An interesting question regards the effectiveness of the destroy and repair methods: are they all useful in the quest for a good solution? If some method is only rarely producing a good solution, would it be better to disable it and invest the corresponding computation time into the other methods? We devised two experiments to answer these questions.

In the first experiment we checked that all the methods were being used when producing good solutions. We ran the tuned algorithms on all instances and, each time a new best solution was produced, we recorded which method was used. Since at the beginning of the solution process it's much easier to improve on the best solution, we skipped the first 1000 iterations from the count. With did this with the same setup used for tuning: short runs of 5 minutes, and long ones of 30 minutes. This small experiment proved that, indeed, all methods were being used, but in different proportions. For example, the statistics relative to the 30-minute runs are presented in Figure 9. We can see how the situation is more balanced for destroy methods, while for repair methods the greedy one gets more than 80% of the hits; this is in line with what has been observed for ALNS algorithms regarding the effectiveness of "aggressive" repair methods (see, e.g., (Santini et al., 2018, Section 2)).

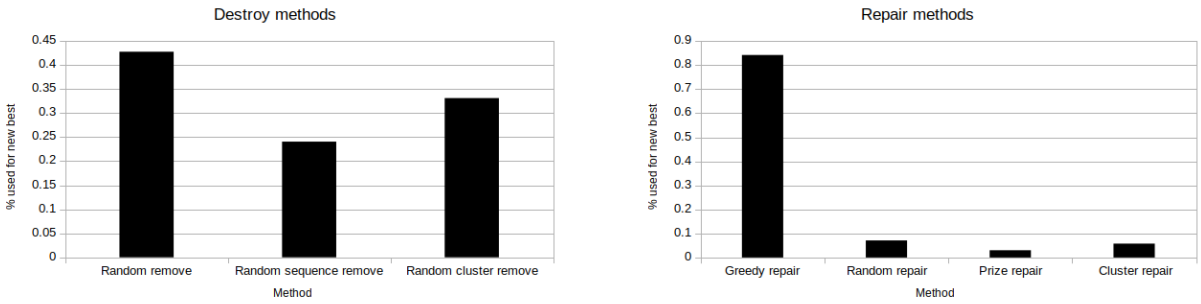


Figure 9: Number of times each destroy (left) and repair (right) method was used to produce a new best solution during the long (30 minutes) runs.

	B&C			GRASP			Iterative			Genetic			ALNS				
Instances	**G%**	**T**	**NB**	**G%**	**T**	**NB**	**G%**	**T**	**NB**	**G%**	**T**	**NB**	**G%**	**T**	**TB**	**NB**	**BKS**
Gen1 Medium	0.00	248.05	45	0.66	7.66	32	2.15	**1.14**	21	0.62	2.12	30	**0.14**	99.51	7.28	**40**	0
Gen2 Medium	0.00	92.89	45	1.19	2.48	19	0.76	**1.92**	11	0.63	2.38	14	**0.15**	173.77	57.68	**29**	0
Gen3 Medium	0.00	149.66	45	0.96	4.18	17	1.69	**1.80**	16	0.90	2.31	15	**0.58**	177.83	41.78	**30**	0
Gen4 Medium	0.00	837.30	45	0.41	2.96	17	0.65	**1.83**	15	0.38	2.09	18	**0.17**	51.93	7.46	**25**	0
Tot Medium	0.00	331.98	180	0.81	4.32	85	1.31	**1.67**	63	0.63	2.23	77	**0.26**	125.76	28.55	**124**	0
Gen1 Large	4.01	7433.41	16	5.73	7893.56	0	8.58	1329.29	1	2.38	**990.82**	7	**1.15**	11802.68	9117.14	**18**	2
Gen2 Large	3.54	11644.10	13	7.21	4389.82	0	6.30	2198.50	0	1.31	**1093.37**	11	**1.06**	12827.93	11225.54	**18**	7
Gen3 Large	6.33	13749.78	17	6.52	4814.36	0	5.27	2082.26	0	1.12	**1109.93**	9	**0.77**	12796.79	10436.91	**16**	8
Gen4 Large	4.05	17087.41	20	5.51	3807.94	1	5.42	1987.85	1	**1.03**	**767.54**	**14**	1.13	9256.99	7393.82	10	8
Tot Large	4.48	12478.68	66	6.24	5226.42	1	6.39	1899.48	2	1.46	**990.42**	41	**1.03**	11671.10	9543.35	**62**	25
Total	2.24	6405.33	246	3.52	2615.37	86	3.85	950.57	65	1.05	**496.32**	118	**0.64**	5898.43	4785.95	**186**	25

Table 4: Comparison of the four algorithms reported in (Kobeaga, Merino & Lozano, 2018a) and the ALNS presented in this paper, run with a long time limit (5 hours). Columns "G%" indicate gaps, columns "T" average running times, columns "NB" the number of best solutions found, column "TB" the time to produce the best solution, and column "BKS" the number of new best known solutions found.

We then proceeded repeating the runs deactivating in turn each method. A method was deemed removable if the version of the algorithm with the method deactivated passed a Wilcoxon test (Wilcoxon, 1945) against the base version. The test gave interesting results: it indicated that we can obtain better results with the 30-minute time limit if we disable the *prize repair* method, and with the 5-minute time limit if we disable the *greedy repair* method. The reason why disabling *greedy repair* on a shorter time limit improves the results is probably that it is (by large) the most time consuming of the four repair methods. During a 5-minute run it is more profitable to do more iterations and a thorough local search step, than using *greedy repair*. For the final computational tests, therefore, we disabled *prize repair* for the 5-hour runs, and *greedy repair* for the 5-minute runs.

5.2.3 Computational results

We start by presenting the results of the 5-hour runs, which we directly compare to those presented by Kobeaga et al. (2018a). Aggregate results are presented in Table 4, while complete results are available at (Santini, 2018). The results for "B&C", "GRASP", "Iterative" and "Genetic" are copied from the tables published in (Kobeaga et al., 2018a). Columns "G%" report the average gap between the best solution produced by that algorithm, and the best solution (i.e., the best solution produced by any of the five algorithms). Columns "T" report the average computation time over 10 reruns for the four algorithms which were run multiple times, or over a single run for B&C. For ALNS, we show in column "TB" the time used to find the returned solution (i.e., the last time x^* was updated) averaged over 10 reruns. Columns "NB" report the number of instances for which the algorithm produced a solution with the same objective value as the best solution. For ALNS, we show in column "BKS" the number of new best known solutions it found. When computing the average times of B&C, we omitted from the calculations those instances for which the programme crashed, as reported by Kobeaga et al. (2018a). Analogously, when computing the average gaps of GRASP, we omitted those instances for which no solution was produced at the end of the time limit. The last line reports the averages of the numbers above for columns "G%", "T", and "TB", and their sum for columns "NB" and "BKS". Numbers in bold identify the best values across the four heuristic algorithms. We note that the times reported in the table for ALNS do not include the time it takes to tune and run the DBSCAN algorithm. This time, however, is negligible: the maximum recorded value over all graphs is of 1.04s, with an average of 0.05s and a median of 0.003s.

Instances	B&C	GRASP	Iterative	Genetic	ALNS	Genetic + ALNS
Gen1 Medium	45	32	21	30	40	41
Gen2 Medium	45	19	11	14	29	30
Gen3 Medium	45	17	16	15	30	30
Gen4 Medium	45	17	15	18	25	32
Gen1 Large	16	0	1	7	18	25
Gen2 Large	14	0	0	11	18	29
Gen3 Large	17	0	0	9	16	25
Gen4 Large	20	1	1	14	10	24
Total	245	86	65	120	184	236

Table 5: Comparison of the number of best known solutions found by the five algorithm, and by taking the best solution among Genetic and ALNS.

	Genetic				ALNS			
Instances	G%	NB	G*%	NB*	G%	NB	G*%	NB*
Gen1 Medium	0.005	33	0.010	31	0.001	43	0.005	35
Gen2 Medium	0.003	25	0.006	12	0.001	34	0.004	22
Gen3 Medium	0.004	24	0.012	14	0.002	38	0.010	21
Gen4 Medium	0.001	29	0.005	19	0.001	28	0.005	20
Gen1 Large	0.022	27	0.054	4	0.021	16	0.054	0
Gen2 Large	0.039	30	0.062	5	0.016	11	0.041	0
Gen3 Large	0.025	20	0.043	1	0.006	21	0.025	3
Gen4 Large	0.012	30	0.038	4	0.016	12	0.043	1
	0.013	218	0.028	90	0.008	203	0.023	102

Table 6: Comparison of Genetic and ALNS with a 5-minute time limit. Columns "G%" indicate the gap with the best solution obtained with the short time limit, while columns "G*%" report the gap with the best solution obtained with the long time limit. Analogously for columns "NB" and "NB*" which instead refer to the number of best solutions found.

Table 4 prompts a few observations:

- The exact B&C is unbeatable, in terms of solution quality, on medium instances: it consistently finds the optimal solution. On large instances, heuristic approaches are competitive: the lowest gaps are attained by ALNS in generations 1, 2, and 3, and by Genetic in generation 4.
- Genetic produces lower average gaps than ALNS in generation 4, and produces 14 best solutions, compared to the 10 of ALNS. However, out of these 10 solutions, ALNS produced 8 new best known solutions giving tighter lower bounds compared to those existing in the literature. Overall, ALNS produced 25 such new best known solution during the 5-hour runs.
- GRASP and Iterative, while perhaps attractive because of short computing times, produce solutions of inferior quality. This is accentuated in the case of large-size instances.
- Genetic and ALNS are quite complementary in terms of the instances on which they perform better, in particular on the large ones. This is made clear in Table 5, which reports the number of best solutions found by each algorithm (an information also presented in Table 4), but also adds a new column "Genetic + ALNS", which shows what would happen if we were to take the better of the two solutions given by either Genetic or ALNS. It is intersting that, for the large classes, the sets of instances whose best solution is found by Genetic and ALNS are completely disjoint.
- ALNS is able to obtain better solutions but in longer times, as we perhaps expected given the termination criterion used. If a user is interested in simply finding high-quality solutions and has a good time budget available (e.g., solving a tactical or operational problem) he should probably favour ALNS over the other heuristics. If, on the other hand, time is a critical factor (e.g., solving a real-time problem) then Genetic might look as a viable alternative.
- Comparing columns "T" and "TB", we see that on smaller instances ALNS is able to find the best solution in the beginning of the run, and then cycles without improving. On larger instances, it continues improving on the best solution almost until the end of the run.

As mentioned before, we ran additional experiments (only running ALNS and Genetic) to assess the performances of the two algorithms on short runtimes. This time, both algorithms were run on our cluster and the only termination criterion was a time limit of five minutes. We considered for both algorithms the best solution out of 5 reruns. A summary of the results is presented in Table 6, while complete results are available at (Santini, 2018). Columns "G%" report the gap between the algorithm's solution and the best of the two solutions (Genetic and ALNS); columns "G*%", instead, reports the gap with the best solution obtained with the 5-hour time limit. Columns "NB" report the number of times one algorithm produced a better (or equal) solution than the other. Columns "NB*" report the number of times one algorithm produced a result equal (or better) than the best result found with the 5-hour time limit.

We noticed that gap "G*%" was negative — i.e., one of the 5-minute runs produced a better result than the best of the 5-hour runs — 3 times for ALNS, and 12 times for Genetic. The high number for Genetic is perhaps obtained because fixing the time limit to 5 minutes represents sometimes an increase compared to the actual runtime the algorithm had with the more complex termination criterion. In other words, there are cases when at least 25% of the population is "flattened" to the current best individual in less than 5 minutes and, at some later iteration, a better individual is found. Out of the 3 new best known solution (BKS) found by ALNS, one instance coincides with a new BKS the algorithm had found with the longer timeout. Therefore, the total number of BKS found by ALNS is increased from 25 to 27. Of the 12 new BKS found by Genetic, none coincides with a new BKS that ALNS had found with the longer timeout. Therefore, we can attribute to Genetic 12 new BKS.

When limiting the comparison to the results produced in five minutes (columns without asterisk), Genetic produces a slightly higher number of better solutions (218 vs. 203). On the other hand, ALNS has lower average gaps (0.008% vs. 0.013%), hinting that when ALNS returns a solution worse than Genetic's, the quality difference is small; but when Genetic returns a solution worse than ALNS', the quality difference is larger. When we compare the results to those obtained with a longer time limit (columns with the asterisk), ALNS performs better both in terms of average gap and of number of best solutions found.

6 Conclusions

We have presented a new heuristic algorithm for the Orienteering Problem, based on the Adaptive Large Neighbourhood Search paradigm. This is the first time, to the best of our knowledge, that this paradigm is used to solve the OP. We have tested new neighbourhood structures and, in particular, we have explored for the first time the use of a clustering of the input graph to define neighbourhoods via the introduction of the new Random cluster remove and Cluster repair methods. We also introduced a new neighbourhood, Random sequence destroy, based on the removal of entire chunks of the current tour. We tested the algorithm on a standard benchmark library, comparing it with state-of-the-art methods, including the recent algorithm of Kobeaga et al. (2018a) ("Genetic").

The results show that our heuristic ("ALNS") can provide solutions of high quality on long runs, compared to other heuristic methods. The exact algorithm ("B&C"), however, still finds the highest number of best known solutions when given a time limit of 5 hours — despite having a large average gap, while ALNS has the lowest. The true strength of heuristic methods emerges when the available time is limited. Runs with a time limit of 5 minutes show, indeed, that both Genetic and ALNS find good solutions quickly. Genetic finds slightly more best solutions, while ALNS has a slightly lower average gap. In summary, both heuristic methods are competitive when solution time is a critical factor, and they even show a high degree of complementarity, hinting that algorithm selection procedures could be explored.

Future research directions include extending the ALNS algorithm to the case with multiple vehicles (Team Orienteering Problem) and to other popular variants of the OP, such as the the OP with time windows. It would also be interesting to investigate the use of a dynamic clustering algorithm, where the clusters can change during the solution process, based on information collected during past iterations (for example, how many times a vertex appeared in good solutions).

Acknowledgements

The author whishes to thank Eduardo Álvarez-Miranda for the many fruitful discussions on the topics presented in this paper.

References

Aksen, D., Kaya, O., Salman, F. S. & Tüncel, Ö. (2014). An adaptive large neighborhood search algorithm for a selective and periodic inventory routing problem. *European Journal of Operational Research*, *239*(2), 413–426.

Borràs, J., Moreno, A. & Valls, A. (2014). Intelligent tourism recommender systems: A survey. *Expert Systems with Applications*, *41*(16), 7370–7389.

Campos, V., Mart, R., Sánchez-Oro, J. & Duarte, A. (2014). Grasp with path relinking for the orienteering problem. *Journal of the Operational Research Society*, *65*(12), 1800–1813.

Clarke, G. & Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, *12*(4), 568–581.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations research*, *6*(6), 791–812.

Dueck, G. (1993). New optimization heuristics: The great deluge algorithm and the record-to-record travel. *Journal of Computational physics*, *104*(1), 86–92.

Ester, M., Kriegel, H.-P., Sander, J. & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD conference proceedings* (pp. 226–231). AAAI Press.

Feillet, D., Dejax, P. & Gendreau, M. (2005). Traveling salesman problems with profits. *Transportation science*, *39*(2), 188–205.

Fischetti, M., Salazar-González, J. J. & Toth, P. (1998). Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, *10*(2), 133–148.

Golden, B., Levy, L. & Vohra, R. (1987). The orienteering problem. *Naval research logistics*, *34*(3), 307–318.

Gunawan, A., Lau, H. C. & Vansteenwegen, P. (2016). Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, *255*(2), 315–332.

Hansen, P. & Mladenovi, N. (2003). Variable neighborhood search. In *Handbook of metaheuristics* (pp. 145–184). Springer.

Helsgaun, K. (2000). An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, *126*(1), 106–130.

iRace User Guide. (2018). *The iRace package: user guide*. Version 3.1. Retrieved from https://cran.r-project.org/web/packages/irace/vignettes/irace-package.pdf

Kobeaga, G., Merino, M. & Lozano, J. A. (2018a). An efficient evolutionary algorithm for the orienteering problem. *Computers & Operations Research*, *90*, 42–59.

Kobeaga, G., Merino, M. & Lozano, J. A. (2018b). Compass: An orienteering problem solver. Retrieved from https://github.com/bcamath-ds/compass

Kobeaga, G., Merino, M. & Lozano, J. A. (2018c). OPLib: A library of orienteering problem instances. Retrieved from https://github.com/bcamath-ds/OPLib

Kovacs, A. A., Parragh, S. N., Doerner, K. F. & Hartl, R. F. (2012). Adaptive large neighborhood search for service technician routing and scheduling problems. *Journal of scheduling*, *15*(5), 579–600.

Li, Y., Chen, H. & Prins, C. (2016). Adaptive large neighborhood search for the pickup and delivery problem with time windows, profits, and reserved requests. *European Journal of Operational Research*, *252*(1), 27–38.

Liang, Y.-C., Kulturel-Konak, S. & Lo, M.-H. (2013). A multiple-level variable neighborhood search approach to the orienteering problem. *Journal of Industrial and Production Engineering*, *30*(4), 238–247.

Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, *44*(10), 2245–2269.

Lin, S. & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, *21*(2), 498–516.

Lloyd, S. (1982). Least squares quantization in PCM. *IEEE transactions on information theory*, *28*(2), 129–137.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T. & Birattari, M. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, *3*, 43–58.

Marinakis, Y., Politis, M., Marinaki, M. & Matsatsinis, N. (2015). A memetic-grasp algorithm for the solution of the orienteering problem. In *3rd international conference on modelling, computation and optimization in information systems and management sciences* (pp. 105–116). Springer.

Reinelt, G. (1991). TSPLIB - a traveling salesman problem library. *ORSA journal on computing*, *3*(4), 376–384.

Resende, M. & Ribeiro, C. (2005). GRASP with path relinking: Recent advances and applications. In T. Ibaraki, K. Nonobe & M. Yagiura (Eds.), *Metaheuristics: Progress as real problem solvers* (pp. 29–63). Springer.

Ropke, S. & Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, *40*(4), 455–472.

Santini, A. (2018). "orienteering-alns" on GitHub. doi:10.5281/zenodo.1248517

Santini, A., Ropke, S. & Hvattum, L. M. (2018). A comparison of acceptance criteria for the adaptive large neighbourhood search metaheuristic. *Journal of Heuristics*, *In press*.

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

Sevkli, Z. & Sevilgen, F. E. (2010). Discrete particle swarm optimization for the orienteering problem. In *2010 ieee congress on evolutionary computation (cec)*. IEEE.

Silberholz, J. & Golden, B. (2010). The effective application of a new approach to the generalized orienteering problem. *Journal of Heuristics*, *16*(3), 393–415.

Souffriau, W., Vansteenwegen, P., Vertommen, J., Berghe, G. V. & Oudheusden, D. V. (2008). A personalized tourist trip design algorithm for mobile tourist guides. *Applied Artificial Intelligence*, *22*(10), 964–985.

Thomadsen, T. & Stidsen, T. (2003). *The quadratic selective travelling salesman problem*. Danish Technical University. Retrieved from http://www.forskningsdatabasen.dk/en/catalog/2389481607

Tsiligirides, T. (1984). Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, *35*(9), 797–809.

Vansteenwegen, P., Souffriau, W. & Van Oudheusden, D. (2011). The orienteering problem: A survey. *European Journal of Operational Research*, *209*(1), 1–10.

Vansteenwegen, P. & Van Oudheusden, D. (2007). The mobile tourist guide: An OR opportunity. *OR insight*, *20*(3), 21–27.

Wang, X., Golden, B. & Wasil, E. (2008). Using a genetic algorithm to solve the generalized orienteering problem. In B. Golden, R. Raghavan & E. Wasil (Eds.), *The vehicle routing problem: Latest advances and new challenges* (pp. 263–274). Springer.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, *1*(6), 80–83.

Appendix A Things that did not work

We believe that publishing negative results is also important, to save the time of future researchers who might want to build on the current work. For this reason, we list here a series of ideas that we tested, but did not produce a positive effect on the performance of the algorithm.

Better initial solutions. When constructing the initial solution for the ALNS algorithm, other than randomly shuffling the vertices as explained in Section 4.3, we also tried sorting them by prize and by distance to the depot. Furthermore, for each of these three methods, we tested both a simple greedy construction algorithm and a variation where we also improve the solution via local search. The local search consists of applying 2-opt followed by the Greedy repair method. It seems intuitive that at least some of these methods should perform better than the basic random greedy construction. However, it is less clear if starting from a better solution has any impact on the overall performance of the algorithm, because the high number of iterations and the use of very large (and, thus, overlapping) neighbourhoods usually limits the impact of the quality of the initial solution in ALNS algorithms. To answer the first question, we report here a summary of the average solution values obtained by each of the six methods. The values in the table on the left represent the average deviation with the initial solution giving the highest profit. Therefore, a value of 1 denotes the best-performing method and the smaller the value, the worse the method performed. The table on the right reports the p-values of pairwise Wilcoxon tests to assess the significance of the difference in the averages (we deemed significant the differences giving $p \leq 0.05$). In this table, we highlight in bold the tests which gave a negative result, i.e., $p > 0.05$.

Instances	Random	Random + LS	Prize	Prize + LS	Distance	Distance + LS
Gen1 Medium	**1.00**	**1.00**	0.96	0.98	0.90	0.97
Gen2 Medium	0.97	**1.00**	0.94	0.99	0.88	**1.00**
Gen3 Medium	0.99	**1.00**	0.98	0.98	0.87	0.93
Gen4 Medium	0.99	**1.00**	0.98	0.99	0.86	0.96
Gen1 Large	**1.00**	0.98	0.89	0.96	0.84	0.96
Gen2 Large	0.98	**1.00**	0.87	**1.00**	0.86	**1.00**
Gen3 Large	0.98	**1.00**	0.98	0.98	0.89	0.96
Gen4 Large	0.99	**1.00**	0.98	0.99	0.88	0.98

	R	R+LS	P	P+LS	D	D+LS
R		0.030	< 0.001	0.014	< 0.001	< 0.001
R+LS			< 0.001	< 0.001	< 0.001	< 0.001
P				< 0.001	< 0.001	**0.304**
P+LS					< 0.001	0.002
D						< 0.001
D+LS						

The following two tables are similar to the previous ones, but they compare the final results of running the ALNS algorithm starting from the respective initial solution. To illustrate the point, we limit the run to 10 000 iterations as the only termination criterion.

Instances	Random	Random + LS	Prize	Prize + LS	Distance	Distance + LS
Gen1 Medium	**1.00**	0.99	0.98	0.98	0.98	**1.00**
Gen2 Medium	0.99	**1.00**	**1.00**	**1.00**	**1.00**	0.98
Gen3 Medium	0.99	**1.00**	0.98	0.99	0.99	0.98
Gen4 Medium	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
Gen1 Large	**1.00**	**1.00**	0.96	0.98	0.96	0.97
Gen2 Large	0.99	0.98	**1.00**	0.97	0.96	0.97
Gen3 Large	0.98	0.99	0.96	0.98	0.96	**1.00**
Gen4 Large	**1.00**	**1.00**	0.99	0.99	0.97	0.99

	R	R+LS	P	P+LS	D	D+LS
R		0.049	**0.381**	**0.596**	< 0.001	**0.211**
R+LS			0.001	**0.085**	< 0.001	< 0.001
P				**0.480**	< 0.001	**0.930**
P+LS					< 0.001	**0.179**
D						**0.091**
D+LS						

In the table on the left, we can see how the deviations are much more uniform after running ALNS. For most instance classes there are many configurations all finding solutions of the same quality; it is emblematic the case of Medium instances of Generation 4, for which all versions give almost (to 2-digits significance) identical average deviations. In particular we notice that, given a type of initial heuristic, the versions with or without Local Search give comparable results: 4 vs. 5, 3 vs. 2, 2 vs. 3 scores of "1.00" for Random, Prize, and Distance, respectively. Because the 2-opt part of Local Search involves a considerable time investment — in the order of minutes for the larger instances — we would need more noticeable differences to justify their use in time-limited runs. With respect to the greedy method of choice, Random performs slightly better than the other two orderings. The table on the right shows that, for many of the Wilcoxon tests, the results were inconclusive. The only methods which yield differences in results consistently deemed significant are Distance and Random + LS. In general, the p-values obtained are larger than those in the previous table. Given the analysis above, we opted for keeping the basic greedy initial solution generation procedure, based on the random shuffling of customers.

Adaptive destroy methods. One thing we tried was to dynamically increment the value of α^{rr} up to a maximum value $\bar{\alpha}^{\text{rr}}$ to make sure that enough vertices vere being removed during the destroy phase. Every 5000 iterations without improving on the best solution, then, we were increasing α^{rr} by 10%. If a new best solution was found, the parameter was reset to its original value. The maximum $\bar{\alpha}^{\text{rr}}$ was tuned together with the other parameters, but it turned out that the resulting method was performing worse than the original version.

Faster greedy repair. The greedy repair method demands that, for each non-visited vertex, we evaluate all positions where it could be inserted in the tour. We can argue intuitively that a vertex will be usually placed between tour vertices which are close to it, since this produces a minimal increase in travel time. In the same spirit of what Kobeaga et al. (2018a) proposed for their node insertion heuristic, we associated to each vertex a small list of neighbours (vertices at a short distance) and, when listing all possible insertions we first checked among the neighbours. Only if none of them was already part of the tour, we resorted to a full search (in contrast to Kobeaga et al. (2018a) which aborted the search in this case). The only parameter here is the number of vertices to include in the list. We tried with several values (1, 2, 3, 5, 10, 15, 20, 25, 50) but we could not see any difference in the overall performance of the algorithm: no configuration could beat any of the others in a Wilcoxon test. We also tried the approach of Kobeaga et al. (2018a) of only checking insertions adjacent to neighbours without resorting to a full search. All the new configurations but one showed differences deemed not significant (via Wilcoxon tests) compared to their previous counterparts. The only configuration which gave a significant difference ($p \leq 0.05$) was "3 neighbours", but the average gaps were slightly worse when using the approach of Kobeaga et al. (2018a); in other words, the difference was significant but negative. For this reason, we turned off this feature.

Tabu moves for greedy repair. Another idea we tried was to add tabu moves to the greedy reapair algorithm, to prevent it from cycling through the same solutions too often. Every time we inserted a vertex i_2 between other two vertices i_1, i_3, we added the pairs (i_1, i_2) and (i_2, i_3) to the tabu list for 5000 iterations. In subsequent calls to the greedy repair method, we would not insert i_2 immediately after i_1, nor immediately before i_3. This method, however, decreased the quality of the solutions found by the overall algorithm.

Randomised greedy repair. When performing greedy repair, we always choose the best insertion. Another possibility is to list the, say, 10 best insertions and choose one at random among them. We tried this approach, but the results were dramatically worse than going with the "pure greedy" strategy.

Restoring feasibility exactly. When restoring the feasibility of a time-infeasible tour, we attempted to use an exact procedure. The meaning of *exact* is that it should return the highest-prize tour, once the order of visit of the vertices is fixed (as it is given by the order of visit in the infeasible tour). In other words, it should give the optimal set of vertices to drop from the infeasible tour, to maximise the prize collected at the remaining vertices. This optimal set is easy to obtain with, e.g., a labelling algorithm. Consider an infeasible tour $(i_0, \ldots, i_k)$ and consider an auxiliary digraph with vertices $\{i_0, \ldots, i_k, i_{k+1}\}$, where $i_0 = i_{k+1}$ is the depot. The arc set includes all arcs (i_k, i_l) such that $l > k$. A choice of vertices to keep corresponds to a path from i_0 to i_{k+1}. If we associate to each partial path a label (t, p) where t is the total travel time and p the total prize collected along the path, we can easily see that a path with longer travel time and lower prize is always dominated by paths with shorter travel time and higher prize. Considering then, all non-dominated paths to i_{k+1} and choosing the one with the highest collected prize, gives an optimal way to perform feasibility repair. It turned out, however, that this method was no better than the simpler one we already implemented, in terms of the overall best solution found by ALNS.