TU Delft

Delft University of Technology

Adaptive Iterated Local Search with Random Restarts for the Balanced Travelling Salesman Problem

Pierotti, Jacopo; Ferretti, Lorenzo ; Pozzi, Laura ; van Essen, J. Theresia

DOI
10.1007/978-3-030-68520-1_4

Publication date
2021

Document Version
Final published version

Published in
Metaheuristics for Combinatorial Optimization

Citation (APA)
Pierotti, J., Ferretti, L., Pozzi, L., & van Essen, J. T. (2021). Adaptive Iterated Local Search with Random Restarts for the Balanced Travelling Salesman Problem. In S. Greco, M. F. Pavone, E.-G. Talbi, & D. Vigo (Eds.), *Metaheuristics for Combinatorial Optimization* (pp. 37-56). (Advances in Intelligent Systems and Computing; Vol. 1332). Springer. https://doi.org/10.1007/978-3-030-68520-1_4

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Green Open Access added to TU Delft Institutional Repository

'You share, we take care!' - Taverne project

https://www.openaccess.nl/en/you-share-we-take-care

Otherwise as indicated in the copyright section: the publisher
is the copyright holder of this work and the author uses the
Dutch legislation to make this work public.

Adaptive Iterated Local Search with Random Restarts for the Balanced Travelling Salesman Problem

Jacopo Pierotti[1(✉)], Lorenzo Ferretti[2], Laura Pozzi[2], and J. Theresia van Essen[1]

[1] Delft Institute of Applied Mathematics, Delft University of Technology, Delft, The Netherlands
{j.pierotti,j.t.vanessen}@tudelft.nl

[2] Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland
{lorenzo.ferretti,laura.pozzi}@usi.ch

Abstract. Metaheuristics have been widely used to solve NP-hard problems, with excellent results. Among all NP-hard problems, the Travelling Salesman Problem (TSP) is potentially the most studied one. In this work, a variation of the TSP is considered; the main differences being, edges may have positive or negative costs and the objective is to return a Hamiltonian cycle with cost as close as possible to zero. This variation is called the *balanced* TSP (BTSP). To tackle this new problem, we present an adaptive variant of the iterated local search metaheuristic featuring also random restart. This algorithm was tested on the MESS2018 metaheuristic competition and achieved notable results, scoring the 5th position overall. In this paper, we detail all the components of the algorithm itself and present the best solutions identified. Even though this metaheuristic was tailored for the BTSP, with small modifications its structure can be applied to virtually any NP-hard problem. In particular, we introduce the *uneven reward-and-punishment* rule which is a powerful tool, applicable in many contexts where fast responses to dynamic changes are crucial.

Keywords: Iterated Local Search · Travelling Salesman Problem · Balanced Travelling Salesman Problem · Hamiltonian cycle · Metaheuristic

1 Introduction

In the Travelling Salesman Problem (TSP), a salesman has to visit a given set of cities and, after travelling along all of them, has to return to the one he started from, hence, completing a cycle. Given a set of cities V and a cost matrix D with entries $d_{i,j}$ $(i, j \in V)$ –in the *standard* TSP, costs represent the distance between cities–, the goal of the traveller is to find the cycle of minimum cost covering all the cities. The TSP is an NP-hard problem, and therefore, there are no

© Springer Nature Switzerland AG 2021
S. Greco et al. (Eds.): MESS 2018, AISC 1332, pp. 37–56, 2021.
https://doi.org/10.1007/978-3-030-68520-1_4

known techniques able to provide an optimal solution efficiently. In this work, we consider a variation of the *standard* TSP. In particular, we address the *balanced* TSP (BTSP). In this variant, the entries of the cost matrix can be negative and the goal of the problem is to find the cycle, visiting all the cities, with cost as close as possible to zero. Despite the complexity of these problems, the TSP and its variants are widely studied and used in many different contexts. Typical applications are: clustering [1], vehicle routing [2], computer wiring [3], cutting wallpaper [4], job-scheduling [5], DNA sequencing [6] and pattern-allocation [7] problems. To cope with these problems and their complexity, many approaches have been proposed in the literature. The main discriminant among them is their ability to obtain an optimal solution or to rapidly discover a near-optimal one. In the latter, there is a trade off between the time required to identify the solution and the quality of the solution provided.

Among the different approaches proposed in literature, some of the most relevant are: Genetic Algorithms (GA), Ant Colony Optimisation (ACO), Tabu Search (TS), Adaptive Large Neighbourhood Search (ALNS), Simulating Annealing (SA), Local Search (LS) and Iterated Local Search (ILS). The approach proposed by Juneja et al. [8] exploits the ability of population-based heuristics to search for multiple solutions in each iteration of the algorithm and, by using various combinations of selection, crossover and mutation techniques, to continuously improve the quality of current solutions. Dorigo et al. [9] were the first to introduce the possibility to use ACO, a heuristic algorithm which navigates the solution space by mimicking ants finding food as a group, as a viable strategy to solve the TSP. More recently, Escario et al. [10] have refined this approach introducing different types of agents –specialised ants– and population dynamics to organise the ants' movements. The TS approach proposed by Toth et al. [11] is based on the use of restricted neighbourhoods, allowing to reduce the solution space and leading to a more efficient implementation of candidate strategies proposed for tabu search algorithms. The ALNS heuristic, proposed by Ribeiro et al. [12], is based on the algorithm initially devised by Ropke and Pisinger [13] and extends the large neighbourhood search of Shaw [14] by using destroy and repair methods within the same search process. Differently from the previous metaheuristics, SA techniques mimic the metal annealing process by considering probabilistic moves depending on a temperature parameter. The SA methodologies proposed in [15] and [16] have been adopted for the TSP resulting in a viable alternative to the above mentioned metaheuristics. Lastly, LS and ILS techniques have been vastly used to solve the TSP, such as in [17,18] and [19]. As described by Lourenço et al. in [20], by using LS, a sequence of viable solutions is iteratively generated within the embedded heuristic.

In this work, we propose a variation of the ILS technique. ILS, differently from LS techniques, alternates the search phase with the perturbation phase in order to escape tenacious local optima. Given a graph, our approach searches for a Hamiltonian cycle within it and, by iterating over a sequence of actions operating on the current solution, navigates the neighbourhood of the current solution while searching for local optima. Perturbations are then applied in order

to escape local optima, hence searching different regions of the solution space. The main contribution of this work consists in the introduction of an *uneven reward-and-punishment* adaptation rule, which in turn leads to a more reactive response to the current solution.

The paper continues as follows: Sect. 2 presents the problem formulation, Sect. 3 describes the advantages of ILS techniques and introduces as proposed methodology an Adaptive Iterated Local Search with Random Restarts (AILS-RR). Section 4 shows experimental evidence for the effectiveness of the AILS-RR technique introduced, and finally, Sect. 5 concludes the paper and presents final remarks.

2 Problem Formulation

The goal of the TSP is to minimise the cost of the cycle connecting a set of given cities that a salesman has to visit exactly once. The cycle of the salesman can be defined as a sub-graph of graph $G(V, E)$ where V is the set of vertices –cities– and E is the set of edges of the graph representing the connections between cities. Given two cities $i, j \in V$, we define $d_{\overline{ij}}$ as the cost of travelling from city i to city j. In addition, we define a binary variable $x_{\overline{ij}}$ which specifies if the cycle of the salesman includes the edge from city i to city j as follows:

$$x_{\overline{ij}} = \begin{cases} 1, & \text{if edge } \overline{ij} \in E \text{ is in the cycle of the salesman} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Then, we define the total cost of the cycle as:

$$C_{total} = \sum_{\overline{ij} \in E} d_{\overline{ij}} x_{\overline{ij}}. \tag{2}$$

In the *classic* TSP, all costs are strictly positive and the objective is to find the Hamiltonian cycle of minimum cost. In the BTSP, costs can be either positive or negative and the objective is to find the Hamiltonian cycle of cost as close as possible to zero. A Hamiltonian cycle is a connected sequence of edges that joins a sequence of vertices, such that each vertex in V is visited exactly once and the edge sequence is closed. A closed sequence starts from a vertex i and, through an ordered sequence of vertices connected by edges, returns to the original vertex i. Given an ordered sequence, we name adjacent vertices of a generic vertex its previous and its following vertex. In general, graphs are not fully connected, i.e. not all vertices $i, j \in V$ are connected by a direct edge. Edges are undirected and each of them is associated with a cost which can be either positive or negative. We define the degree of a vertex as the number of its outgoing edges.

The differences between the BTSP and the TSP translate the model of the *classic* TSP in a similar one, hereby reported in (3)–(8):

$$\min C_{total} \tag{3}$$

$$C_{total} \geq \sum_{\overline{ij} \in E} d_{\overline{ij}} x_{\overline{ij}} \tag{4}$$

$$C_{total} \geq -\sum_{\overline{ij} \in E} d_{\overline{ij}} x_{i,j} \tag{5}$$

$$\sum_{j \in V} x_{\overline{ij}} = 2 \qquad \forall i \in V \tag{6}$$

$$\sum_{i,j \in T, i \neq j} x_{\overline{ij}} \leq |T| - 1 \qquad \forall T \subset V, T \neq \emptyset \tag{7}$$

$$x_{\overline{ij}} \in \{0, 1\} \qquad \forall \overline{ij} \in E \tag{8}$$

$$C_{tot} \in \mathbb{R}^{+} \tag{9}$$

Objective function (3) and Constraints (6), (7) and (8) are standard TSP constraints. In particular, Constraints (6) impose the cycle to visit each vertex exactly once –one incoming, one outgoing edge– while Constraints (7) avoid the presence of sub-tours T in the cycle. Constraints (8) force variables $x_{\overline{ij}}$ to be binary and Objective function (3) aims at minimising the absolute cost of the cycle. Lastly, considering that is a minimisation problem, Constraints (4), (5) and (9) impose variable C_{tot} to assume the absolute value of the cost of the cycle. By minimising the absolute value, we force the costs to be as close to zero as possible.

To solve this problem, we used AILS-RR, which is variant of ILS. We define as a feasible solution any Hamiltonian cycle and as objective function the cost of the cycle itself. Thus, we aim at identifying the best candidate in the neighbourhood of the current solution by applying modifications to the solution structure.

3 Methodology

The Iterated Local Search methodology repeatedly builds a sequence of solutions generated by a local search heuristic embedded in a framework. As defined by Lourenço et al. in [20], given a current solution s, ILS generates an intermediate solution $\hat{s}$ by applying changes on s. Then, the embedded local search heuristic is applied to $\hat{s}$, which leads to a new solution s'. If this solution improves s, it becomes the new current solution in our sequence, and thus we keep navigating the solution space modifying s'. Otherwise, if s' does not introduce an improvement with respect to s, ILS continues to apply modification to s. Starting from a feasible solution, ILS explores its neighbourhood and determines the best solution within it. Thus, this metaheuristic exploits the possibility to search for a solution in a reduced space defined by the output of the local search heuristic, instead of searching over the entire solution space.

The methodology proposed in this paper, i.e. AILS-RR, relies on ILS by searching in the neighbourhood of an already existing solution. In order to apply AILS-RR, it is necessary to generate a first solution to be fed to the AILS-RR

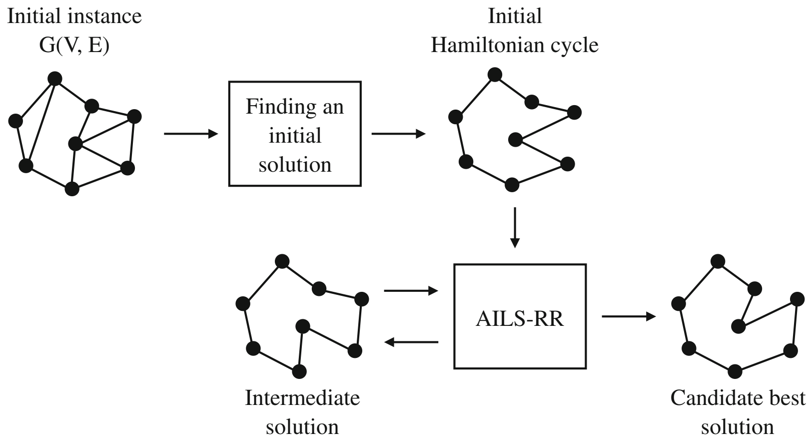

Fig. 1. The AILS-RR framework and its phases.

itself. Our methodology starts generating an initial solution from a graph $G(V, E)$ representing a BTSP instance. Once a solution has been identified, this is given as input to AILS-RR, which iteratively searches for improving solutions until a certain stopping criterion is reached. Figure 1 shows an overview of the steps of our methodology.

Details about the strategy adopted to identify an initial solution are presented in Sect. 3.1, while the AILS-RR procedure is described in Sect. 3.2.

3.1 Finding an Initial Solution

Determining whether there exists a Hamiltonian cycle in a not fully connected graph is an NP-complete problem [21]. In order to find a Hamiltonian cycle, we used the Snakes-and-Ladders Heuristics (SLH) by Baniasadi et al. [22]. The SLH is a polynomial time algorithm inspired by the k-opt heuristic. In SLH, vertices are ordered on a circle where edges between adjacent vertices represent the arcs of the circle while all other edges are considered as chords of the circle. The heuristic attempts to place all edges of a Hamiltonian cycle on the circle by seeking changes in the arrangement of vertices of the graph, with the goal of maximising the number of edges on the circle.

For this work, we have relied on the online implementation of SLH [23] provided by the authors of [22]. However, the online implementation accepts a maximum of 2000 vertices. To generate a cycle in the instances with more than 2000 vertices, we randomly partitioned all the vertices in equally sized subsets, with size smaller than 2000. For each of them, we independently found a Hamiltonian cycle; then, considering only edges from one subset to another, we selected the k vertices with highest degree. Finally, for each of the k vertices and its adjacent ones in the Hamiltonian cycle, namely A and B, we checked if they

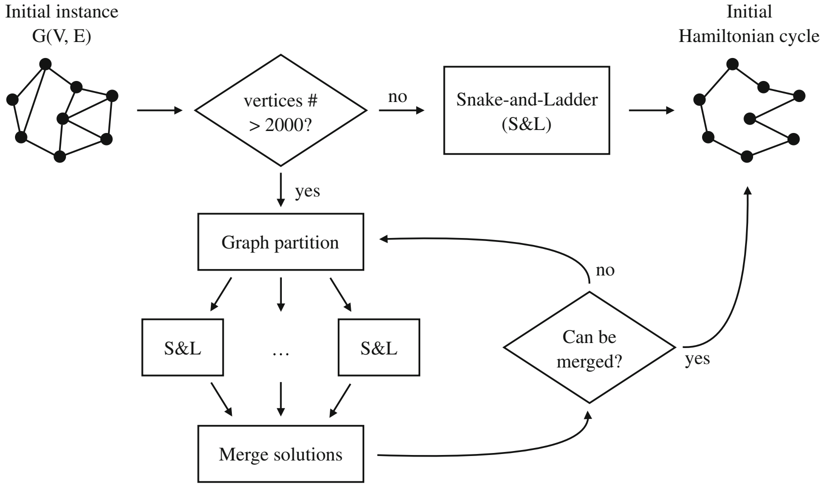

Fig. 2. Flowchart of the process to generate the initial solution.

were connected to any couple of adjacent vertices in the other subset, namely C and D. If so, it means that the two disjoint cycles can be united as $A-$*first cycle*$-B-C-$*second cycle*$-D-A$. When none of the k vertices could be used to join the two cycles, the whole procedure was repeated by randomly partitioning the nodes. Figure 2 shows a flowchart of the process used to identify an initial solution. For all the evaluated instances, we were able to find an initial solution without having to re-partition more than two times.

3.2 Adaptive Iterated Local Search with Random Restarts

Once an initial feasible solution is found, we used AILS-RR to improve its objective value. The AILS-RR procedure proposed in this work relies on four main phases: *local search*, *update*, *perturbation* and *random restart*.

The procedure starts from the initial solution s provided by the method described in Sect. 3.1. Then, a *local search* is performed on s and a solution s' is returned. This solution can be: a) a new solution different from s or b) the same solution s in case no improving solution has been identified. Thus, the *update* phase amends the local search heuristic according to the resulting solution generated. In this *update* phase, the adaptive part of the algorithm takes place.

Once solution s' has been generated and the *update* has been performed, we replace s_{best} – the best solution found so far – with s' if the latter strictly improves it. In case no improved solution, with respect to s, was found after $MaxIteration$ consecutive iterations, a *perturbation* to s is applied and a new solution is generated. Then, the next AILS-RR iteration starts from the

perturbed solution and the counter for the not improving solution is reset. Additionally, if for *too many* consecutive iterations, no solution has improved s_{best}, a *random restart* from a *good* known solution is performed. In particular, details on the number of consecutive iterations needed and on what is considered a *good* solution are presented in Sect. 3.2.4. Random restart is applied to avoid extensively searching unpromising regions of the solution space, hence moving to a more fruitful one. Finally, the AILS-RR terminates whenever the stopping criterion is met. Further details on the stopping criterion are given in Sect. 3.2.5.

A pseudo-code of the approach is illustrated in Algorithm 1. In the algorithm, there are five main functions: a) *LocalSearch*, b) *Update*, c) *Perturbation*, d) *RandomRestart* and e) *StoppingCriterion*. Our algorithm differentiates from the *standard* ILS [20] in terms of the update phase as well as the introduction of the random restart from a known solution. In the following paragraphs, we explain each component of the algorithm in detail.

Algorithm 1. Adaptive Iterated Local Search with Random Restart

1: **procedure** AILS (Input: Graph G, Hamiltonian cycle s; Output: Hamiltonian cycle s_{best})
2: $s_{best} = s$;
3: $notImproving = i = 0$;
4: **while** not *StopCriterion()* **do**
5: $i = 0$;
6: **while** $i <$ MaxIterations **do**
7: s'=*LocalSearch*(s);
8: *Update(LocalSearch())*;
9: **if** *Cost*(s') $<$ *Cost*(s) **then**
10: $s = s'$;
11: $i = 0$;
12: **if** $|Cost(s')| < |Cost(s_{best})|$ **then**
13: $s_{best} = s'$;
14: $notImproving = 0$;
15: **else**
16: $notImproving++$;
17: **end if**
18: **else**
19: $i++$; $notImproving++$;
20: **if** $notImproving > RestartFactor$ **then**
21: s = *RandomRestart()*;
22: $notImproving = 0$;
23: **end if**
24: **end if**
25: **end while**
26: s=*Perturbation*(s);
27: **end while**
28: return s_{best}
29: **end procedure**

3.2.1 Local Search

Starting from a current solution s, the local search aims at finding an improved solution s'. It consists of applying modifications, which are dictated by different operators, to the solution structure. An operator is a function that, given cycle s, applies modifications on its structure which generates multiple cycles which are variations of cycle s. The resulting cycles define a neighbourhood of s. Every solution in the neighbourhood is evaluated and only the solution which most improves s is accepted. If no solution improves s, s itself is returned. During the local search, the algorithm uses – with probability depending on their weights – one of these three operators: *one edge insertion*, *two edges insertion* and *cycle modification*. Each operator selects at least one edge to be inserted in s. The insertion of an edge divides the original cycle in two subtours. Selecting the edge to be inserted determines which subtours will be created. Dually, identifying a desired subtour lets us establish which edge is to be inserted. Given the particular structure of the instances, we have chosen to determine the edges first. More on this is presented in Sect. 4.1. The following paragraphs introduce the operators adopted.

One Edge Insertion. Given a cycle s, the first operator selects, at random, one edge $\overline{AB}$ which is not in s. The extreme points of the edge, A and B, are adjacent to two vertices each in s –C and D for A, E and F for B. For the time being, we assume every vertex to be different with respect to each other; straightforward modifications can be applied if this is not the case. There is only a limited number of possibilities to insert edge $\overline{AB}$ in the existing solution; at most, there are eight possible outcomes. The cost, Eq. (3), of all possible outcomes is evaluated and the best one is chosen. Since the graph is not complete, in general not all the combinations exist. Indeed, naming p the probability that an edge exists, and assuming they are all independent, we can analyse quantitatively the probability for each combination to exist. Figure 3 shows all possible outcomes; the edge to be inserted is represented in blue, the edges that may or may not exist are shown in red, while black indicates the edges belonging to the original cycle. By construction, we know the existence of edges $\overline{AB}$, $\overline{AC}$, $\overline{AD}$, $\overline{BE}$ and $\overline{BF}$. Hence, we deduce there are two combinations with probability p, four combinations with probability p^2 and two combinations with probability p^3.

Two Edges Insertion. Similarly to the previous operator, this process chooses two edges not yet in the current solution and tries to insert them. If the four extreme vertices of the two selected edges are all different, isolating them divides the cycle in four subtours. Hence, the solution is now decomposed in six subtours – four from the original cycle and two from the two inserted edges, that can be considered subtours as well, see Fig. 4ii.

There are 10!![1] possible ways to combine the four subtours and the two edges. This number comes from $(2 \cdot (t-1))!!$, where t is the number of subtours – six, in our case – and -1 because a degree of freedom is lost for the intrinsic symmetry

[1] !! is double factorial, i.e. $f!! = f \cdot (f-2) \cdot (f-4)...$
In our case, $10!! = 3840$.

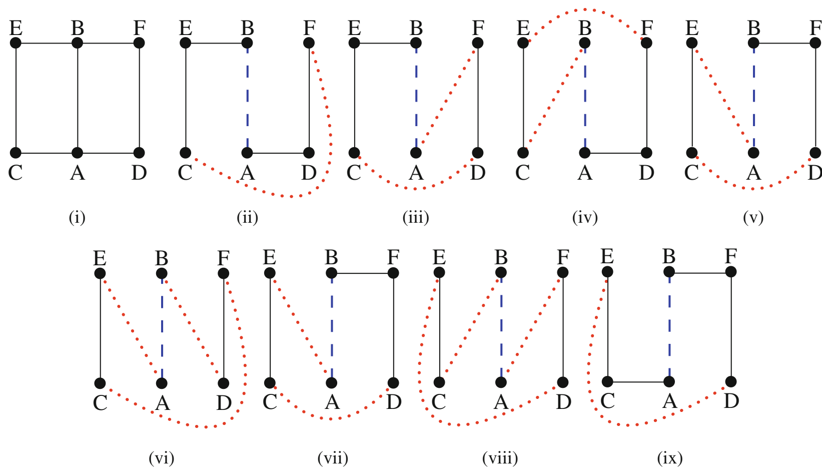

Fig. 3. Examples of single edge insertion. Dashed blue lines show the edge to be inserted; dotted red lines indicate the edges that may or may not exist, while black lines show the edges belonging to the original cycle. Figure (i) shows the original cycle, while figures (ii)–(ix) show the possible insertions.

of cycles. A multiplicative factor of 2 is added since each subtour can be linked to the next one through two different endpoints. Having t subtours implies having, as their endpoints, $2t$ vertices. Intuitively, the double factorial follows because a vertex can be connected to $2t-2$ other vertices, every vertex but itself and the other endpoint of its subtour. Once connected, the following vertex can be connected to $2t-4$ others. This includes all the vertices but itself, the other endpoint of its subtour and the endpoints of the subtour to which it is already linked to. Recursively, we can see how this develops, for the remaining vertices, in a double factorial structure. Among these combinations, only the ones with at least probability p^3 to exist are considered by our methodology.

Generally speaking, these first two operators can be viewed as modified versions of k-opt. Figure 4 shows an example of two edges insertion. Starting from an initial cycle – Fig. 4i – two edges are inserted. The new edges divide the cycle into four different subtours – Fig. 4ii. Finally, Fig. 4iii and Fig. 4iv show an example of a reconstructed cycle with probability p^3 and p^6, respectively.

Cycle Modification. The two edge insertion generates multiple intermediate solutions, but it is computationally more expensive with respect to the one edge insertion operator. To compensate the computational requirements of the two edge insertion operator, we introduce the cycle modification operator.

This operator selects, at random, an edge in the existing solution s. We name A and B its extreme vertices, which are consecutive in the original solution s. Then, we select at random one edge, not in solution s, which is outgoing A and is entering, without loss of generality, in C. At the same time, we select at random one edge, not in solution s, which is outgoing B and is entering,

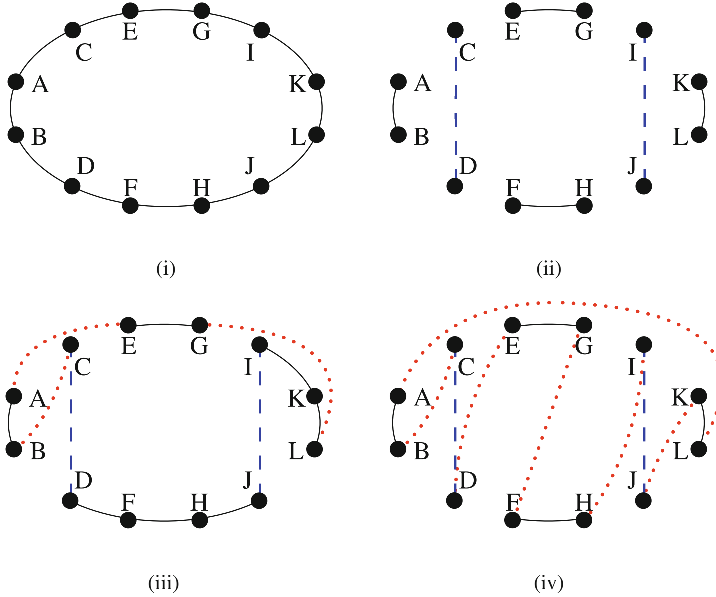

Fig. 4. Example of two edge modification. (i) Initial cycle. (ii) Insertion of two edges and subtours generated. (iii) Reconstructed cycle with probability p^3. (iv) Reconstructed cycle with probability p^6. The original edges are shown in black, while dashed blue lines depict the inserted edges, and dotted red lines depict the edges with probability p.

without loss of generality, in $D \neq C$, see Fig. 5ii. Subsequently, we consider the path, in the original cycle, from C to D, that passes through A and B. In that cycle, we name E and F the follower of C and the predecessor of D, respectively. By construction, there exist paths $\overline{EA}, \overline{CD}, \overline{BF}$ and edges $\overline{AC}, \overline{BD}$. Hence, there exists a path connecting $\overline{EA} - \overline{AC} - \overline{CD} - \overline{DB} - \overline{BF}$, see Fig. 5iii. Finally, if edge $\overline{EF}$ exists, we obtain a feasible cycle, see Fig. 5iv. In general, edge $\overline{EF}$ exists with probability p. To increase the size of the neighbourhood, this procedure is repeated for all outgoing edges of B. It is not, however, repeated for all combinations of outgoing edges of A and outgoing edges of B, because this would be computationally too expensive.

3.2.2 Update

Each operator of the local search is applied with a probability proportional to its associated weight. These weights are constrained to be greater than a parameter $MinWeight$ and their sum is forced to a value lower that the upperbound parameter $MaxWeights$. Whenever an operator returns a solution which does not improve the input solution, we subtract f –in this work, f has value 1– from its associated weight. In case an operator returns a better solution than the solution given as input, its associated weight is increased by 10% and rounded

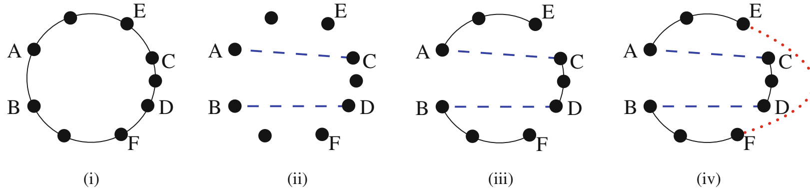

Fig. 5. Example of cycle modification. (i) Original cycle. (ii) Selection of the edges to be inserted. (iii) Construction of the existing path. (iv) Closing the cycle with edge $\overline{EF}$ which exists with probability p. Original edges are shown in black, dashed blue lines depict the inserted edges, and dotted red lines indicate the edges that exist with probability p.

to the nearest higher integer. In addition, if the returned solution is even better than the best known solution $-s_{best}-$ the weight of the operator leading to the improved solution receives an extra reward of $10f$ in addition to the normal reward obtained for improving the previous solution. We call this discrepancy among a constant decrease and a proportional increase an *uneven reward-and-punishment* adaptation rule. In our opinion, an even reward-and-punishment adaptation rule is more suited to grasp stable characteristics, such as the ones related to the structure of the *graph* itself, while an uneven rule is more keen to quickly adapt to variations, such as the ones in the changing structure of the *solution*.

3.2.3 Perturbation

The *perturbation* is applied when we are not able to improve a local solution for a significant number of iterations $-MaxIterations$. To perform a perturbation, the algorithm uses the same operators as the local search. The main differences, with respect to the local search, is that every change is accepted –not only an improving one– and it is performed only once. There is no evidence that more perturbations results in better solutions. In fact, more perturbations cause the current solution to drift too much away from a promising part of the solution space. In addition, since the costs of the edges are neither Euclidean nor the authors found any pattern within them, even a slight modification of a few edges can lead to dramatic changes in the objective function.

3.2.4 Random Restart

Perturbations allow to explore different regions of the solution space; nonetheless, some of those regions could be unpromising. To avoid exploring inadequate regions of the solution space, it is useful to restart the search from a region where *good* solutions are known to exist. If, after *too many* consecutive iterations, no solution improved the best known objective function, then a *random restart* from a *good* known solution is performed. In particular, we define as $History$ an array storing the $HistorySize$ best solutions and their number of occurrences. If, after

$RestartFactor$ consecutive not improving iterations, s_{best} was not improved, we perform a random restart from any of the solutions stored in $History$.

We define $RestartFactor$ as:

$$RestartFactor = cMax + \frac{MostVistitedSolution(History)}{HistoryStep}, \quad (10)$$

where $MostVistitedSolution(History)$ assumes the value of the number of visits to the most visited solution in $History$, while $cMax$ and $HistoryStep$ are parameters. $cMax$ indicates the minimum number of iterations the algorithm has to perform before a *random restart* can happen, while $HistoryStep$ is a scaling factor. While we perform the random restart to avoid going too far away from a region of the solution space where good solutions exist, we reduce the frequency of restarts when the same solution is visited more and more times to escape that tenacious local minimum. In fact, it could happen that too frequent restarts drives the local search to the same local minima. In addition, restarting from any of the solutions stored in $History$ helps to maintain a certain degree of diversity.

3.2.5 Stopping Criterion

AILS-RR has no memory of all the solutions discovered since it started, and in general there is no guarantee of optimality. Hence, without a stopping criterion, it would indefinitely search for improving solutions. The stopping criterion we implemented terminates the execution of the algorithm if any of the following conditions is met: a) the solution cost is zero, and thus, we have reached the optimal solution, b) a user-defined time limit was exceeded, c) the algorithm returned for more than $MaxIterationHistory$ times the same solution.

If condition a) is met, then, it is not possible to further improve the solution identified. Condition b) offers a knob for setting a reasonable usage of resources required to search for improving solutions, and condition c) is useful to avoid expensive explorations of particularly tenacious local minima from where the algorithm cannot escape even with its *perturbation* move.

4 Experiments

In this section, we explain the experiments setup and the performance of our algorithm. In Sect. 4.1, we describe the instances tested, in Sect. 4.2, the parameters used, and lastly, in Sect. 4.3, the performance of our algorithm.

4.1 Instances

The algorithm was tested on 27 given instances, available at [24], which vary in size from 10 vertices and 40 edges, to 3,000 vertices and 12,000 edges. Hence, on average, each vertex has degree 4. This motivates the analysis on the probability of existence of an edge in the AILS-RR. The absolute value of the costs of every

edge can be written as $k_1 \cdot 100{,}000 + k_2$, where k_1 and k_2 are integers in the range $[0, 99]$. We run the algorithm twice per instance. The first time, we used as input the instances considering as cost only k_1. In the following, we refer to this change in the cost associated with the edges as a cost modification. With this data, the algorithm was able to find a solution of cost zero for all instances. Then, the algorithm was run a second time, starting from the previously found solution, with the real costs of the arcs.

4.2 Parameters Tuning

The local search procedure is repeated until for $MaxIterations = 100$ consecutive iterations no improving solution is found. $HistorySize$, the number of how many *good* solutions were stored in array $History$, is set to 100. Parameters $cMax$ and $HistoryStep$, which are used to determine when to restart from a random solution in $History$, are set to 1,000 and 100, respectively. Parameter $MaxIterationHistory$ determines how many times a solution can be visited before the stopping criterion is met and is set to 1,000,000. This means that a random restart can happen as often as after 1,000 consecutive not improving iterations, or as rarely as after 10,999 consecutive not improving iterations. In Paragraph 3.2.4, we explained how often the random restart happens depending on how many times the most inspected solution is visited. We may have a restart after 10,999 consecutive not improving iterations and not after 11,000 times as expected if $MostVisitedSolution(History)$ assumes value $MaxIterationHistory$. In fact, $MostVisitedSolution(History)$ cannot assume value $MaxIterationHistory$ in Eq. (10) because, if so, the stopping criterion is met and the execution of the whole algorithm is terminated. Every single operator weight is initially set to 333 and restricted to integer values above $MinWeight = 1$ and such that their sum does not exceed $MaxWeights = 1{,}000$. If the sum of the weights exceeds $MaxWeights$, the weight of every parameter is decreased by one, unless this violates the lower bound $MinWeight$, until the threshold is respected. For the tests with the modified cost, the maximum running time for each instance was set to two hours while, for the tests with the original cost, the maximum running time for each instance was set to twelve hours. Table 1 summarises all the parameters used in the algorithm.

4.3 Performance

Instances were run overnight on different machines. In particular, instances up to 100 nodes were run on an Intel Core i7-6600U CPU @2.60 GHz 2.80 GHz with 8 GB RAM and instances from 150 to 400 nodes on a Intel Core i7 @2.9 GHz with 8 GB RAM. Bigger instances (500 to 3,000 nodes) were run on a 32 core machine with Intel Xeon E5-4650L CPU @2.60 GHz 3.1 GHz with 500 GB of physical memory. Since the BTSP is a new problem, introduced for the MESS2018 solver challenge, no comparison with the state of the art is possible. In general, optimal solutions are not known but they cannot have a better objective function than zero. By executing the AILS-RR on the instances with modified costs of the

Table 1. Parameters tuning

Parameter name	Value	Description
MaxIterations	100	Maximum not improving cycles of LS
HistorySize	100	Dimension of array *History*
cMax	1,000	Minimum number of cycles for a random restart
HistoryStep	100	Random restart parameter
MaxIterationHistory	1,000,000	Stopping condition parameter
MinWeight	1	Minimum weight for each operator
MaxWeights	1,000	Maximum value of the sum of the weights of all operators
MaxTime	2 h	Maximum time per instance - modified costs
MaxTime	12 h	Maximum time per instance - original costs

edges, as explained in Sect. 4.1, we know that the optimal solutions for all these modified instances have cost zero. In general, a zero cost solution for the modified instances does not translate into an optimal solution for the original instances; nonetheless, it is a *good* initial when solving the instance with original costs. The results displayed in Sect. 4.3.1 and 4.3.2 refer to the modified costs and the results shown in Sect. 4.3.3 refer to the original costs. For the tests with the modified costs, we set a time limit of two hours and a limit of 10,000 iterations by counting how many times *LocalSearch* is called. We ran this experiment to perform a qualitative analysis of the obtained results and operator effectiveness. We tested the instances on a 32 core machine with Intel Xeon E5-4650L CPU @2.60 GHz 3.1 GHz with 500 GB of physical memory, and each test was run 10 times in order to obtain average results. In Sect. 4.3.1, we introduce in detail a meaningful instance case, while in Sect. 4.3.2, we present results for all instances.

4.3.1 Instance 3,000 Vertices

The instance presented in this section is representative of the entire set. In fact, Fig. 6i and Fig. 6ii display the trends of the objective function, for the same executions, with respect to iterations and time while Fig. 6iii and Fig. 6iv show the evolution of the weights during the ten tests of the algorithm, with respect to iterations and time. Since all the tests are plotted simultaneously, these figures give some idea on the variance of the trends and how many tests terminated their execution in just a few iterations. First of all, plotting results with respect to iterations or with respect to time only slightly modifies the overall shape of the figures. This is due to the fact that comparable amounts of time are needed for each operator to perform its local search. In Fig. 6iii and Fig. 6iv, sharp peaks with slow decline are visible. This is exactly the effect of the *uneven reward-and-punishment adaptation rule*; since increases are proportional while decreases are constant, rapid changes in the weights of the operators are visible. In this case, it is clear that operator *cycle modification* performs better than the others; as

shown in Sect. 4.3.2, this is the case for basically all other instances. Secondly, Fig. 6i and Fig. 6ii show the absolute value of the best solution found so far by the algorithm. Even though this instance is the biggest one, even for the worst of the ten tests, our algorithm was able to find an optimal solution in roughly two minutes.

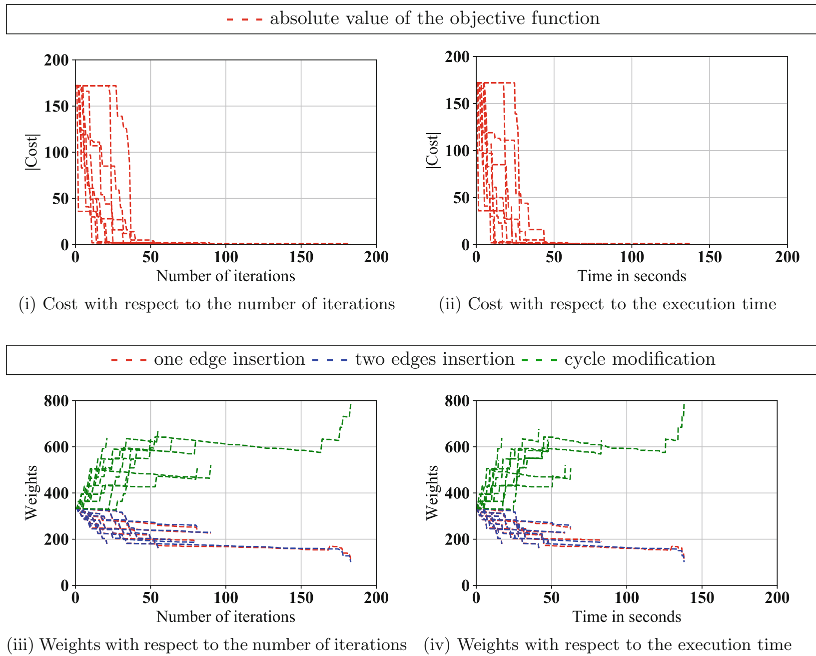

Fig. 6. Evolution of absolute cost of the solutions and operator weights with respect to number of iterations and execution time. Results are shown for the instance with 3,000 vertices.

4.3.2 Results for All Instances

Small instances were solved in a few iterations with no particularly interesting trend; because of that, in this paragraph, we consider only instances of size strictly greater than one hundred vertices. Since no particular difference arises from plotting with respect to the number of iterations or with respect to time, the figures in this paragraph refer to the iterations. Furthermore, for the sake of readability, we decided to plot average results instead of all the 10 trends. Averaging the results highlights the trends but smooths peaks which instead are visible in Fig. 6iii and Fig. 6iv. For instances with more than one hundred nodes, trends of the weights of the operators and of the solution developments are shown in Fig. 7 and Fig. 8, respectively. These trends show the average results for the ten tests. Figure 7 shows that, for all simulations, all tests over all the instances, but one, returned the optimal solution within few iterations –resulting

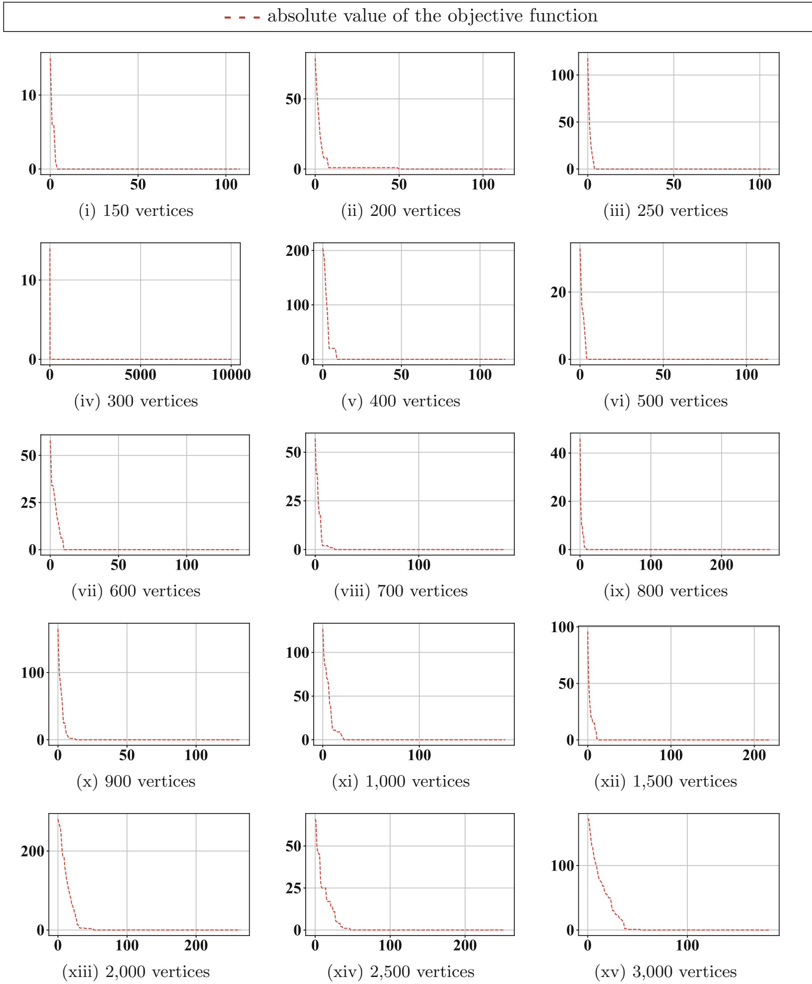

Fig. 7. Evolution of the objective function over different instances. Number of iterations on the x-axis, objective function value on the y-axis.

in few minutes of execution time–, way before encountering the time or the iteration limit. In our opinion, this is a powerful indicator of the effectiveness of our algorithm. Similarly, Fig. 8, shows how among all the instances, the *cycle modification* is the most effective operator. Nonetheless, it is worth noticing that, while for the medium-sized instances, weights are *almost* equivalently distributed among operators, the bigger the instance, the greater the probability of *cycle modification* to be chosen.

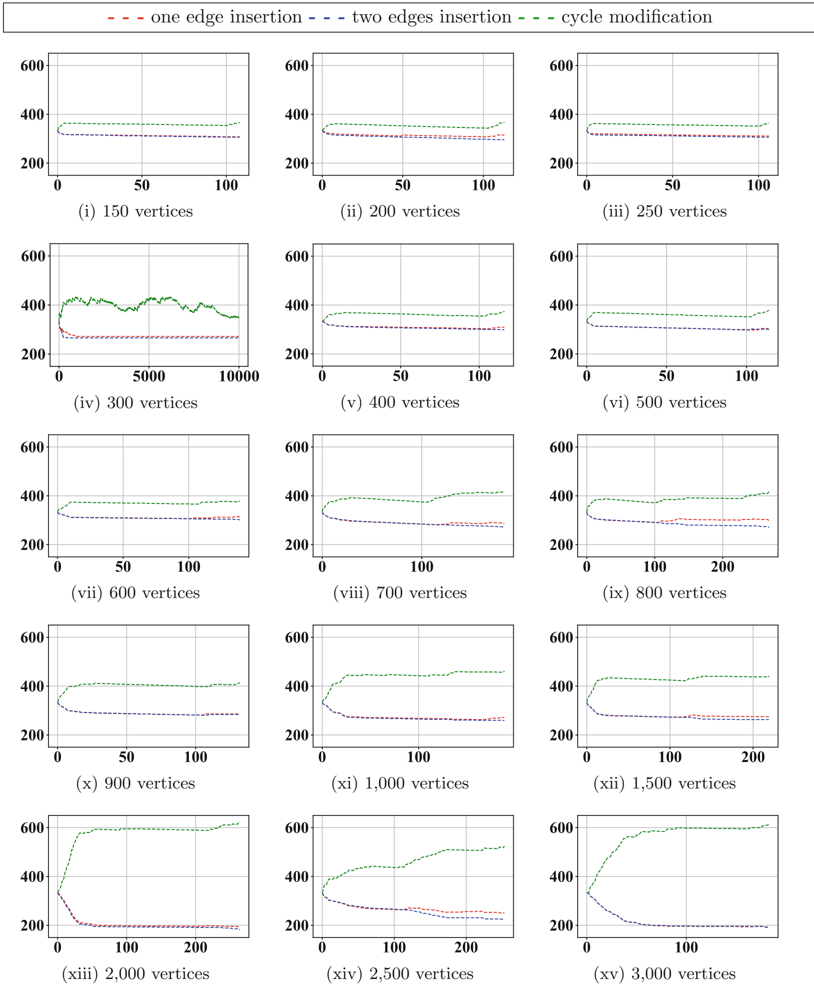

Fig. 8. Evolution of the weights assigned to the different operators over different instances. Number of iterations on the x-axis, objective function value on the y-axis.

4.3.3 Final Results

In this section, for the sake of further comparison, we display the results submitted to the competition. All the results proposed in this section are computed with the original costs. In particular, Table 2 portraits: in the first column, the instance size –expressed in the number of vertices–, and in the second column, the absolute value of the solutions.

Table 2. Results.

# vertices	|Solution cost|
10	105
15	271
20	296
25	375
30	433
40	473
50	717
60	918
70	1,056
80	929
90	1,098
100	1,245
150	2,035
200	2,657
250	3,811
300	4,846
400	6,509
500	8,418
600	9,784
700	17,989
800	18,233
900	20,596
1,000	22,597
1,500	37,662
2,000	49,882
2,500	36,607
3,000	24,423

5 Conclusion

This paper illustrates the AILS-RR methodology applied to the balances travelling salesman problem. With slight modifications of the local search operators, we believe that the same metaheuristic can obtain significant results in many operational research problems. The proposed metaheuristic is a variant of ILS and it features the adaptive use of the local search operators and restart moves. Key advantages of the AILS-RR are: its effectiveness in navigating the solution space, as shown in the achieved ranking in the MESS2018 Metaheuristics Competition, its easiness to implement and its ability to quickly obtain near optimal

solutions. Additional motivations and a detailed description of our algorithm are presented in Sect. 3, which presents the algorithm structure focusing on the different phases of the metaheuristic. In particular, the description details the main contribution of the proposed methodology which lays in the newly introduced *uneven reward-and-punishment adaptation* rule. To the best of our knowledge, this is the first time that such a strategy is used. Section 4 proves that our AILS-RR achieves notable results, scoring remarkable positions in almost every instance ranking, and achieving the 5th position in the competition.

Acknowledgement. The authors want to thank the organisers of the MESS2018 summer school for the challenging opportunity they offered us. The first author wants to acknowledge the DIAMANT mathematics cluster for partially funding his work.

References

1. Lenstra, J.K.: Clustering a data array and the traveling-salesman problem. Oper. Res. **22**(2), 413–414 (1974)
2. Lenstra, J.K., Kan, A.R.: Complexity of vehicle routing and scheduling problems. Networks **11**(2), 221–227 (1981)
3. Lenstra, J.K., Kan, A.R.: Some simple applications of the travelling salesman problem. J. Oper. Res. Soc. **26**(4), 717–733 (1975)
4. Hahsler, M., Hornik, K.: TSP-infrastructure for the traveling salesperson problem. J. Stat. Softw. **23**(2), 1–21 (2007)
5. Whitley, L.D., Starkweather, T., Fuquay, D.: Scheduling problems and traveling salesmen: the genetic edge recombination operator. In: 3rd International Conference on Genetic Algorithms, vol. 89, pp. 133–140 (1989)
6. Caserta, M., Voß, S.: A hybrid algorithm for the DNA sequencing problem. Discret. Appl. Math. **163**, 87–99 (2014)
7. Madsen, O.B.: An application of travelling-salesman routines to solve pattern-allocation problems in the glass industry. J. Oper. Res. Soc. **39**(3), 249–256 (1988)
8. Juneja, S.S., Saraswat, P., Singh, K., Sharma, J., Majumdar, R., Chowdhary, S.: Travelling salesman problem optimization using genetic algorithm. In: 2019 Amity International Conference on Artificial Intelligence, pp. 264–268. IEEE (2019)
9. Dorigo, M., Gambardella, L.M.: Ant colonies for the travelling salesman problem. Biosystems **43**(2), 73–81 (1997)
10. Escario, J.B., Jimenez, J.F., Giron-Sierra, J.M.: Ant colony extended: experiments on the travelling salesman problem. Expert Syst. Appl. **42**(1), 390–410 (2015)
11. Toth, P., Vigo, D.: The granular tabu search and its application to the vehicle-routing problem. Inf. J. Comput. **15**(4), 333–346 (2003)
12. Ribeiro, G.M., Laporte, G.: An adaptive large neighborhood search heuristic for the cumulative capacitated vehicle routing problem. Comput. Oper. Res. **39**(3), 728–735 (2012)
13. 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)
14. Shaw, P.: A new local search algorithm providing high quality solutions to vehicle routing problems. APES Group, Department of Computer Science, University of Strathclyde, Glasgow, Scotland, UK (1997)

15. Geng, X., Chen, Z., Yang, W., Shi, D., Zhao, K.: Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. Appl. Soft Comput. **11**(4), 3680–3689 (2011)
16. Malek, M., Guruswamy, M., Pandya, M., Owens, H.: Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. Ann. Oper. Res. **21**(1), 59–84 (1989)
17. Johnson, D.S.: Local optimization and the traveling salesman problem. In: International Colloquium on Automata, Languages, and Programming, pp. 446–461. Springer (1990)
18. Voudouris, C., Tsang, E.: Guided local search and its application to the traveling salesman problem. Eur. J. Oper. Res. **113**(2), 469–499 (1999)
19. Paquete, L., Stützle, T.: A two-phase local search for the biobjective traveling salesman problem. In: International Conference on Evolutionary Multi-Criterion Optimization, pp. 479–493. Springer (2003)
20. Lourenço, H.R., Martin, O.C., Stützle, T.: Iterated local search. In: Handbook of Metaheuristics, pp. 320–353. Springer (2003)
21. Garey, M., Johnson, D., Tarjan, R.: The planar Hamiltonian circuit problem is NP-complete. SIAM J. Comput. **5**(4), 704–714 (1976)
22. Baniasadi, P., Ejov, V., Filar, J.A., Haythorpe, M., Rossomakhine, S.: Deterministic "Snakes and Ladders" Heuristic for the Hamiltonian cycle problem. Math. Program. Comput. **6**(1), 55–75 (2014)
23. Snake and Ladders Heuristic. http://www.flinders.edu.au/science_engineering/csem/research/programs/flinders-hamiltonian-cycle-project/slhweb-interface.cfm. Accessed 01 Aug 2018
24. MESS Competition. https://195.201.24.233/mess2018/home.html. Accessed 01 Aug 2018