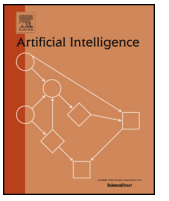
Artificial Intelligence 319 (2023) 103918

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint



The first AI4TSP competition: Learning to solve stochastic routing problems ☆

Yingqian Zhang [a,*,1], Laurens Bliek [a,1], Paulo da Costa [a,1], Reza Refaei Afshar [a,1], Robbert Reijnen [a,1], Tom Catshoek [b,1], Daniël Vos [b,1], Sicco Verwer [b,1], Fynn Schmitt-Ulms [c,2], André Hottung [d,2], Tapan Shah [e,2], Meinolf Sellmann [f,2], Kevin Tierney [d,2], Carl Perreault-Lafleur [g,2], Caroline Leboeuf [g,2], Federico Bobbio [g,2], Justine Pepin [g,2], Warley Almeida Silva [g,2], Ricardo Gama [h,2], Hugo L. Fernandes [i,2], Martin Zaefferer [l,2], Manuel López-Ibáñez [j,2], Ekhine Irurozki [k,2]

[a] *Eindhoven University of Technology, Netherlands*
[b] *Delft University of Technology, Netherlands*
[c] *McGill University, Canada*
[d] *Bielefeld University, Germany*
[e] *General Electric, USA*
[f] *InsideOpt, USA*
[g] *Université de Montréal, Canada*
[h] *Polytechnic Institute of Viseu, Portugal*
[i] *Rockets of Awesome, New York City, USA*
[j] *University of Manchester, UK*
[k] *Telecom Paris, France*
[l] *DHBW Ravensburg, Germany*

ARTICLE INFO

Article history:
Received 25 January 2022
Received in revised form 31 October 2022
Accepted 31 March 2023
Available online 3 April 2023

Keywords:
AI for TSP competition
Travelling salesman problem
Routing problem
Stochastic combinatorial optimization
Surrogate-based optimization
Deep reinforcement learning

ABSTRACT

This paper reports on the first international competition on AI for the traveling salesman problem (TSP) at the International Joint Conference on Artificial Intelligence 2021 (IJCAI-21). The TSP is one of the classical combinatorial optimization problems, with many variants inspired by real-world applications. This first competition asked the participants to develop algorithms to solve an orienteering problem with stochastic weights and time windows (OPSWTW). It focused on two learning approaches: surrogate-based optimization and deep reinforcement learning. In this paper, we describe the problem, the competition setup, and the winning methods, and give an overview of the results. The winning methods described in this work have advanced the state-of-the-art in using AI for stochastic routing problems. Overall, by organizing this competition we have introduced routing problems as an interesting problem setting for AI researchers. The simulator of the problem has been made open-source and can be used by other researchers as a benchmark for new

☆ This paper was submitted to the Competition Section of the journal.

* Corresponding author.
E-mail address: yqzhang@tue.nl (Y. Zhang).

[1] The organization team.
[2] The winning teams.

https://doi.org/10.1016/j.artint.2023.103918

learning-based methods. The instances and code for the competition are available at https://github.com/paulorocosta/ai-for-tsp-competition.

1. Introduction

Many real-world optimization problems are combinatorial optimization problems (COPs) with the objective to find an optimal solution among a finite set of possible solutions. COPs are proven to be NP-Complete, thus solving them to optimality is computationally expensive and mostly impractical for large instances. COPs have been studied extensively in various research communities, including discrete mathematics, theoretical computer science, and operations research. An efficient way of finding acceptable solutions for COPs is through heuristic approaches. The time complexity of heuristics is mainly polynomial, although they may provide solutions that are far from optimal. Besides, these approaches must be redesigned if the problem assumption and settings are changed. Recent years have seen rapidly growing interest in using machine learning (ML) to dynamically learn heuristics and find close-to-optimal solutions for COPs [1]. Among COPs, routing problems such as the traveling salesman problem (TSP) are well-known, and they emerge in many real-life applications. The TSP has several variants that include uncertainty, making the problem challenging for traditional exact and heuristic algorithms. TSP and its variants are some of the most well-studied COPs in the ML literature. Previous works on deep neural network approaches for routing problems have focused on learning to construct good tours [2–13] and on learning to search for good solutions [14–27], leveraging supervised and deep reinforcement learning (DRL). Other approaches considered surrogate-based optimization (SBO) [28–33], using ML models to guide the search for good tours.

In this competition, the participants solve a variant of TSP using ML methods. The selected variant of TSP contains stochastic weights, where the cost of traveling between two nodes is stochastic. Each node also has a prize, and collecting the prize depends on the arrival time of an agent. These assumptions make this variant of TSP similar to real-life problems. For example, in real life, the required time to travel from one city to another depends on road construction work and traffic jams. Moreover, visiting a location is usually assigned with time bounds that must be respected. To solve this problem variant, the participants must use one of two ML methods: SBO or DRL. Both of these methods have shown considerable promise in generating solutions for routing problems in previous works.

We emphasize that the primary goal of this competition is to bring new surrogate-based and DRL-based approaches into practice for solving a difficult variant of TSP. This is done by attracting ML researchers and challenging them to solve this difficult routing problem. The solutions may be built upon existing work adapted for the particular TSP variant. Although some previous work has focused on prize collecting (orienteering) problems or stochastic weights, few researchers take the combination of these assumptions into account. This motivates us to establish a platform that provides the opportunity for AI researchers to develop SBO and DRL approaches for solving a well-known routing problem. As a byproduct, the competition provides several winning methods and a simulator for generating problem instances that researchers can use to benchmark their ML-based approaches. In summary, the objective of organizing this competition is threefold: (1) to introduce routing problems as an interesting problem setting for ML researchers; (2) to advance the state-of-the-art in using ML for routing problems; and (3) to provide a challenging problem and a simulator for researchers to benchmark their ML-based approaches.

We divide the competition into two tracks, each requiring different knowledge from sub-fields of AI:

- Track 1 (SBO): Given one instance, previously tried tours, and the total reward (sum of the prizes collected in a tour) for those tours, the goal is to learn a model predicting the reward for a new tour. Then an optimizer finds the tour that gives the best reward according to that model, and that tour is evaluated, giving a new data point. Then the model is updated, and this iterative procedure continues for a fixed number of steps. Over time, the model becomes more accurate, giving better and better tours. This procedure is used in SBO algorithms such as Bayesian optimization [34].
- Track 2 (DRL): We consider an environment (simulator) that can generate a set of multiple instances $\mathcal{I}$ following the same generating distribution. We expect as output (partial) solutions containing the order in which the nodes should be visited. The environment returns general instance features and the stochastic travel time for traversing the last edge in a given solution. The goal is to maximize the prizes collected while respecting time-related constraints over multiple samples of selected test instances. This procedure is related to neural combinatorial optimization [4].

The first competition, named the AI4TSP competition, was an IJCAI-21 (International Joint Conference on Artificial Intelligence) competition. It ran from May 27 to July 12, 2021, and was organized by the Delft University of Technology and the Eindhoven University of Technology. By the deadline of the final test phase, we had received four submissions in the SBO track and three submissions in the DRL track. The submissions are tested on up to 1,000 problem instances with up to 200 nodes, and the winners are determined by ranking the total quality of their solutions. The results of the competition have been officially announced in the Data Science Meets Optimization (DSO) workshop, which was co-located with IJCAI-21.

2. Problem description and methodology

Both tracks look at the orienteering problem with stochastic weights and time windows (OPSWTW), which is a simplified version of the time-dependent OPSWTW [35]. This problem is similar to the traveling salesman problem (TSP), where nodes need to be visited while respecting a maximum tour time and opening and closing times of the nodes in order to maximize some measure of rewards. We detail the problem in the section below.

2.1. OPSWTW

In the TSP, the goal is to find the tour with the smallest cost that visits all locations (customers) in a network exactly once. However, in practical applications, one rarely knows all the travel costs between locations precisely. Moreover, there could be specific time windows at which customers need to be served, and certain customers can be more valuable than others. Lastly, the salesman is often constrained by a maximum capacity or travel time, representing a limiting factor in the number of nodes that can be visited.

In this competition, we consider a more realistic version of the classical TSP, i.e., the OPSWTW [35]. In this formulation, the stochastic travel times between locations are only revealed as the salesman travels in the network. The salesman starts from a depot and must return to the depot at the end of the tour. Moreover, each node (customer) in the network is assigned a prize, representing how important it is to visit a given customer on a given tour. Each node has associated time windows. We consider that a salesman may arrive earlier at a node without compromising its prize, but the salesman must wait until the opening time to serve the customer. Lastly, the tour must not violate a total travel time budget while collecting prizes in the network. The goal is to collect the most prizes in the network while respecting the time windows and the total travel time of a tour allowed to the salesman.

Node locations More formally, a OPSWTW problem instance is defined as a set of n nodes that make a complete graph. The x and y coordinates of the nodes in a 2D space are randomly generated integers between two limits. For a node i, the limits are input parameters having the default values of $l_x = \{0, 200\}$ and $l_y = \{0, 50\}$ for x_i and y_i, respectively.

Travel times The noisy travel times $t_{i,j} \in \mathbb{R}$, $\forall i, j \in \{1, \ldots, n\}$ between node i and j are obtained by first computing their Euclidean distance $d_{i,j}$ rounded to the closest integer. Later, this distance is multiplied by a noise term η following a discrete uniform distribution $\mathcal{U}\{1, 100\}$ normalized by a scaling factor $\beta = 100$, i.e., $\tau_{i,j} = d_{i,j}\frac{\eta}{\beta}$, where $t_{i,j}$ are samples from $\tau_{i,j}$. Hence, the travel time between i and j may be different in different samples. Unlike the problem described in [35], the noise term is not time-dependent.

Time windows Each node i has a time window denoted by its lower bound $\{l_i \in \mathbb{N}\}_{i=1}^n$ and upper bound $\{h_i \in \mathbb{N}\}_{i=1}^n$. The time windows are generated around the times to begin service at each node of a second nearest neighbor TSP tour. In more detail, let $d_i^{2\text{nn}}$ be the time of visiting the i-th node in the second nearest neighbor tour, assuming maximum travel times between nodes. The left side of the time window is a randomly generated number between $d_i^{2\text{nn}} - w$ and $d_i^{2\text{nn}}$ where w is an arbitrary integer. Similarly, the right side of the time window is a random number between $d_i^{2\text{nn}}$ and $d_i^{2\text{nn}} + w$. In our problem setting, $w \in \{20, 40, 60, 80, 100\}$ [36].

Prizes Each node has an associated prize. The prize $\{p_i \in \mathbb{R}\}_{i=1}^n$ of node i describes the importance of visiting that node within its time window. The prizes are increasing with the distance between node i and the first node of the tour (depot). In more detail, the prize of each node is determined according to the (rounded) L_2 distances between the nodes and the depot. That is, $p_i = \left(1 + \left\lfloor 99 \cdot \frac{d_{1,i}}{\max_{j=1}^n d_{1,j}} \right\rfloor\right)/100$, where $d_{1,i}$ is the euclidean distance (maximum travel time) from the depot to node i. This prize structure results in challenging instances as it places nodes with higher prizes away from the depot [37].

Constraints Each problem instance has a maximum tour length T that determines the maximum time allowed to be spent on a tour. For each instance, we sample the max tour length T from a discrete uniform distribution $\mathcal{U}\{T_{\min}, T_{\max}\}$, where $T_{\min} = 2 \cdot (\max_{j=1}^n d_{1,j})$ and $T_{\max} = \max(2 \cdot T_{\min}, \lceil \frac{1}{2} d^{\text{nn}} \rceil)$, and d^{nn} is the tour cost of the nearest neighbor TSP solution with maximum travel times. Note that $T_{\min}$ is defined such that it is possible to go from the depot to the farthest node and back. $T_{\max}$ is defined as the maximum between twice the $T_{\min}$ time and half the nearest neighbor TSP tour cost to ensure that instances are still challenging with feasible, but not excessively large values of T. Moreover, solutions must respect the time windows of each node, i.e., $[l_i, h_i]$. That is, if a tour arrives earlier than the opening time of that node, it must wait until the opening time to depart the node. A tour is considered infeasible if the arrival time is higher than the closing time of a node. Note that, unlike the latter case, it is still possible to collect the prizes of a node if arriving earlier than the opening time of the time window.

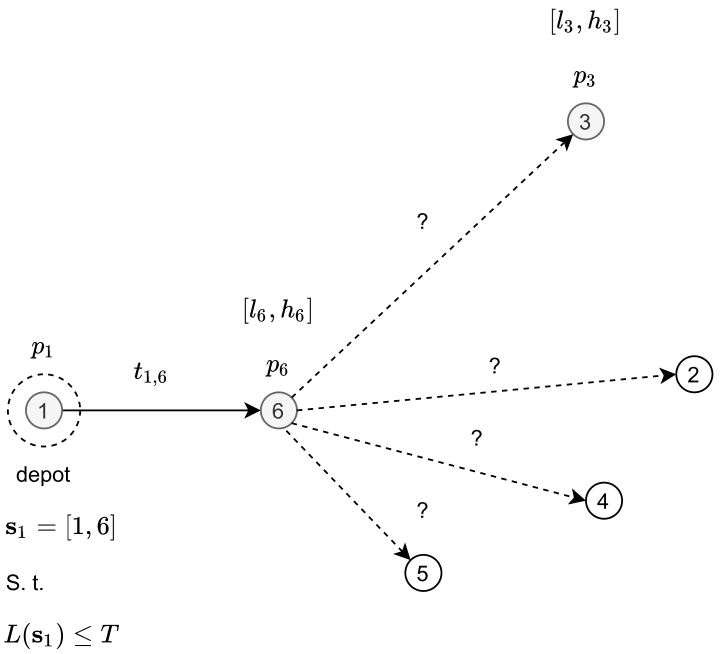


Fig. 1. An instance of the OPSWTW.

Penalties We treat each violation of the constraints of the problem in the form of penalties $\{e_i \in \mathbb{R}\}_{i=1}^{n}$. All solutions (tours) that take longer than T are penalized by $e_i = -n$, incurred at the node i at which the violation first occurred. Moreover, each time window violation from above incurs a penalty of $e_i = -1$ at the current node i at which the violation occurred.

The stochastic travel times, time-dependent constraints, and prizes make this problem difficult to solve with traditional solvers. In our implementation, a problem instance is a complete graph with a particular depot (node:1). The nodes are fixed in a 2D space; however, their travel times are noisy, i.e., they can vary in different runs. The goal of the problem is to find a tour such that the total reward is maximized. Note that prizes, penalties, node coordinates, time window bounds, and the maximum allowed tour time T are known and given in the instance.

To summarize, the main differences between the problem in this competition and the TSP are:

- Not all nodes need to be visited – it is allowed to never visit some nodes.
- Visiting a node after the node's opening time and before its closing time gives a prize.
- Visiting a node after its closing time gives a penalty.
- When visiting a node before its opening time, the agent has to wait until the node opens.
- The time it takes to travel from one node to the other is stochastic.
- The travel times do not directly appear in the objective function.
- The only metric that matters is the sum of collected prizes (penalties).

Fig. 1 shows an example of a next node visitation decision that has to be made by a policy visiting $n = 6$ nodes. In the figure, a tour has visited nodes 1 (depot) and 6 with travel time $t_{1,6}$, which is revealed after visiting node 6. At this current decision moment, we need to choose the next node to visit. The decision should consider the prizes of each node, the time windows, and the total remaining travel time when selecting the next node (in this case, node 3 is selected).

Moreover, when the salesman decides to arrive at a node i earlier than the earliest service time l_i, the travel time gets shifted to the beginning of the time window. For example, if the travel time between the depot (node 1) and node 6 is lower than l_6, the salesman must wait until l_6 to depart from that node. This information becomes available as soon as the salesman arrives at node 6. Lastly, a tour must always return to the depot, and this travel time is also included in the maximum allowed tour time.

2.2. Track 1: surrogate-based optimization

The goal of Track 1 is to solve an optimization problem related to one instance of the OPSWTW problem, finding the tour that maximizes the total reward. The total reward of a tour, which is the sum of all collected prizes and penalties, can be represented as a black-box function $f(\mathbf{s}, I)$, taking as input the instance I and a tour $\mathbf{s}$. The optimization problem is then denoted as:

$$\mathbf{s}^* = \arg\max_{\mathbf{s}} \mathbb{E}[f(\mathbf{s}, I)] \tag{1}$$

for a given instance I. We use the expected value because the simulator is stochastic: it can give different rewards even if the same tour is evaluated multiple times. The expected value for a tour $\mathbf{s}$ is approximated by evaluating $f(\mathbf{s}, I)$ for that tour 10,000 times and calculating the average total reward. This computation takes multiple seconds on standard hardware. Therefore, the problem can be seen as an expensive optimization problem. Surrogate-based optimization methods, such as Bayesian optimization [34], which approximate the expensive objective using online supervised learning, are known to perform well on this type of problem.

The tour $\mathbf{s}$ indicates the order to visit the nodes in the network. It has to take on the specific form $\mathbf{s} = [1, s_1, \ldots, s_n]$, with n the number of nodes and $s_1, \ldots, s_n$ containing all integers from 1 to n. This means that the number 1 will appear twice in the solution. As this number indicates the starting node, the tour consists of starting from the starting node, visiting any number of nodes, then returning to the starting node at some point. Any nodes that appear in the tour after returning to the starting node are ignored.

SBO algorithms approximate the black-box function f in every iteration with a surrogate model g (the online learning problem), then optimize g instead (the optimization problem). Both the results of learning and optimization become better with each simulator call as more data becomes available. The problem is typically split into two sub-problems that are solved every time a tour is given as input to the simulator:

1. Given the tours tried up until now and their corresponding rewards, learn a model to predict how promising any new tour would be.
2. Optimize the model of the previous step to suggest the most promising tour to try next. Then this tour is given as input to the simulator.

The first step can be seen as an online learning problem, where new data comes in at every iteration, and rewards need to be predicted. It also corresponds to the concept of an acquisition function in Bayesian optimization. In step 2, standard optimization methods, such as gradient descent, can be used.

2.2.1. Baseline

As a baseline, we provide an implementation of a standard Bayesian optimization algorithm using Gaussian processes [34]. For this implementation, we use the `bayesian-optimization` Python package [38], after transforming the input space using the approach in [28] and rounding solutions to the nearest integer. The expectation is that such a baseline method does not perform well on the problem due to the combinatorial search space, the many constraints, and the possibility of using noisy, but low-cost approximations of the objective. The competition participants were requested to develop new methods that were more suitable for the problem than existing baseline algorithms.

2.3. Track 2: deep reinforcement learning

In the DRL track, we are interested in a (stochastic) policy π mapping states to action probabilities. A policy in the OPSWTW selects the next node to be visited, given a sequence of previously visited nodes. To cope with the stochastic travel times, the policy must be adaptive. Therefore, the policy needs to consider the instance information to construct tours dynamically that respect the time windows of nodes and the total tour time allowed for the instance. Note that, unlike Track 1, we are interested in general policies applicable to any instance of the OPSWTW in the training distribution.

More formally, we adopt a standard Markov decision process (MDP) $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r \rangle$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}(\mathbf{s}'|\mathbf{s}, a)$ is the transition distribution after taking action a at state $\mathbf{s}$, $r(\mathbf{s}, a)$ is the reward function. Where we model a state $\mathbf{s}$ as partial and complete tours, the action space as the remaining nodes to be visited, and the rewards as the sum of prizes and penalties collected at each step. Thus, the main objective is to find a policy π^* such that

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{I \sim \mathcal{P}(I)} \Big[\mathcal{L}(\pi | I) \Big], \tag{2}$$

where the instance I is sampled from a distribution $\mathcal{P}(I)$ and

$$\mathcal{L}(\pi | I) = \mathbb{E}_{\pi} \left[\sum_{i=0}^{n-1} r(\mathbf{s}_i, a_i) \right], \tag{3}$$

where $\mathbf{s}_i$ is the state at decision epoch i, for example, a partial tour until node s_i, i.e., $\mathbf{s}_i = [s_0, s_1, \ldots, s_i]$, assuming that we always start from the depot, i.e., $s_0 = 1$. Note that the reward after taking action a_i is given by $r(\mathbf{s}_i, a_i) = p_{a_i} + e_{a_i}$ if the tour has not returned to the depot, and 0 otherwise.

2.3.1. Baseline

We provide a baseline to the RL track based on neural combinatorial optimization [4]. Note that this approach is not adaptive and may not perform well in the given task as it only uses the coordinates and prizes to make decisions on

complete tours. Moreover, it is not tailored for stochastic problems as it does not consider the online travel times and maximum tour budget information while traversing the tours. Participants were requested to develop new methods better suited for exploiting the complete information from the instances, including the travel times, revealed at each new location in the tour and the maximum travel budget.

While classical heuristic approaches that do not make use of machine learning might work well for simple problems, we expect new methods based on DRL to be able to outperform them for this problem. The reason for this is that DRL can dynamically learn heuristics that decide which node to visit next based on previously visited nodes, while classical approaches design these decisions by hand. If enough data and computation time for learning is available, these learning capabilities could be leveraged to improve upon hand-tailored heuristics.

3. Technical setup and evaluation

3.1. Problem instances

A set of problem instances was generated and provided for the participants. Each problem instance contains n nodes in 2D space. Each node has an x-coordinate, a y-coordinate, lower and upper bounds of its time window, prizes, and the maximum tour time. The source code of the competition was provided to participants. Thus, everyone had access to how instances and environments were generated, and participants could inspect each of its components. The implementation of the source code of the competition was done using the Python programming language.

In more detail, the `InstanceGenerator` (Algorithm 1) class generates the features of the problem instances. This class is initialized by the number of nodes, limits of x and y coordinates, w, and a random seed. The problem instances are generated in three steps. First, the coordinates of the nodes are generated randomly between two intervals l_x and l_y. Note that the location of the nodes is fixed and given as input; however, the travel times between nodes are subject to change according to noisy values.

Algorithm 1 `InstanceGenerator`.

Require: number of nodes n; limits l_x, l_y; time window size w
  for $i = 1, \ldots, n$ **do**
    $x_i \sim \mathcal{U}\{l_x\}$
    $y_i \sim \mathcal{U}\{l_y\}$
  end for
  $\mathbf{x} \leftarrow [x_1, \ldots, x_n]$, $\mathbf{y} \leftarrow [y_1, \ldots, y_n]$
  $\mathbf{l}, \mathbf{h}, \mathbf{D} \leftarrow$ `TWGenerator`$(\mathbf{x}, \mathbf{y}, w)$
  $\mathbf{p}, T \leftarrow$ `PrizeGenerator`$(\mathbf{x}, \mathbf{y}, \mathbf{D})$
  $I = [\mathbf{x}, \mathbf{y}, \mathbf{l}, \mathbf{h}, \mathbf{p}, T, \mathbf{D}]$
  return I

Second, the time window of each node is generated around the time of visiting that node in the second nearest neighbor tour in the `TWGenerator` class (Algorithm 2). An instance of this class receives the node coordinates of a OPSWTW instance and a value w, computes the L_2 distances between nodes, and returns the time windows for each node based on the second nearest neighbor TSP tour.

Algorithm 2 `TWGenerator`.

Require: node coordinates $\mathbf{x} = [x_1, \ldots, x_n]$, $\mathbf{y} = [y_1, \ldots, y_n]$; time window size w
  for $i = 1, \ldots, n$ **do**
    for $j = 1, \ldots, n$ **do**
      $d_{i,j} \leftarrow L_2(x_i, y_i, x_j, y_j)$
    end for
  end for
  $\mathbf{D} \leftarrow [d_{1,1}, \ldots, d_{n,n}]$
  $\mathbf{s}^{2\text{nn}}, \mathbf{d}^{2\text{nn}} \leftarrow$ `GetSecondNearestNeighbor`$(\mathbf{D})$ ▷ 2nd NN tour and its time (dist.) node-by-node
  $t_1 \leftarrow 0$, $l_1 \leftarrow 0$
  $h_1 \leftarrow \lceil \max_{j=1}^{n} d_j^{2\text{nn}} + w \rceil$
  for $j = 2, \ldots, n$ **do**
    $i \leftarrow \mathbf{s}_j^{2\text{nn}}$
    $l_i \sim \mathcal{U}\{\max(0, d_j^{2\text{nn}} - w), d_j^{2\text{nn}}\}$
    $h_i \sim \mathcal{U}\{\max(0, d_j^{2\text{nn}}), d_j^{2\text{nn}} + w\}$
  end for
  $\mathbf{l} \leftarrow [l_1, \ldots, l_n]$
  $\mathbf{h} \leftarrow [h_1, \ldots, h_n]$
  return $\mathbf{l}, \mathbf{h}, \mathbf{D}$

Third, the prize of each node is determined according to the L_2 distances between the nodes and the depot. The prizes are generated upon calling the `PrizeGenerator` class (Algorithm 3). This class takes as input the node coordinates and

Table 1
A sample problem instance with 4 nodes.

CUSTNO	XCOORD	YCOORD	TW_LOW	TW_HIGH	PRIZE	MAX_T
1	47	24	0	285	0.0	256
2	38	15	102	198	0.19	256
3	53	49	9	52	0.38	256
4	116	23	30	137	1.0	256

the L_2 distance between nodes, i.e., the maximum travel time, and outputs prizes based on the distance between nodes and the depot. That is, nodes farther from the depot have larger prizes. Moreover, this class generates the maximum allowed tour budget of T, preventing tours from obtaining a very big total prize. Participants also had access to the L_2 distance between nodes, which corresponds to the maximum possible travel times between nodes in our setup.

Algorithm 3 `PrizeGenerator`.

Require: node coordinates $\mathbf{x} = [x_1, \ldots, x_n]$, $\mathbf{y} = [y_1, \ldots, y_n]$; maximum time matrix $\mathbf{D}$
$d^{\text{nn}} \leftarrow$ `GetNearestNeighborCost`$(\mathbf{D})$ ▷ d^{nn} is the total travel time of the NN tour
$T_{\min} = 2 \cdot (\max_{j=1}^{n} d_{1,j})$
$T_{\max} = \max(2T_{\min}, \lceil \frac{1}{2} d^{\text{nn}} \rceil)$
$T \sim \mathcal{U}\{T_{\min}, T_{\max}\}$
for $i = 1, \ldots, n$ **do**
$p_i = \frac{\left(1 + \left\lfloor 99 \cdot \frac{d_{1,i}}{\max_{j=1}^{n} d_{1,j}} \right\rfloor\right)}{100}$
end for
$\mathbf{p} \leftarrow [p_1, \ldots, p_n]$
return $\mathbf{p}, T$

As a simple example, assume that there are 4 nodes in an instance, each with a time window, a prize, and a maximum travel budget. An illustration of this example is shown in Table 1. Each row of this table corresponds to a particular node, and the columns are defined as follows: CUSTNO (i) is an integer identifier for the nodes. XCOORD and YCOORD (x_i, y_i) are the coordinate of a node. TW_LOW and TW_HIGH (l_i, h_i) are the left sides and right sides of the time window for each node. PRIZE (p_i) is the prize of each node. Finally, MAX_T (T) is the maximum travel budget. A possible tour for this example is $1 \to 2 \to 3 \to 4 \to 1$. The total time of this tour is 187, and the time of visiting nodes 1, 2, 3 and 4 are 0, 13, 50, and 118, respectively. At time 13, a tour visits node 2; however, it needs to wait until time 102 to collect the prize of this node. If the tour leaves node 2 after collecting its prize, it gets to node 3 after its time window. Therefore, it misses the prize of node 3 and incurs a penalty of -1. Then, it can get to node 4 within its time window and collect its prize. Therefore, the total collected prize for this tour is 0.19.

3.2. Environments

3.2.1. Track 1

The environment for Track 1 (`Env`) receives as input the instance information, containing the node coordinates, time windows, prizes, maximum allowed travel time and the maximum travel times between nodes. The environment implements a method `check_solution` that takes as input a complete tour, i.e., starting and ending at the depot, returning the total reward and tour time of that tour.

3.2.2. Track 2

The environment for Track 2 (`EnvRL`) serves a similar purpose as the one from Track 1. Here, however, participants can interact with the environment on a node-by-node basis. That is, the method `step` expects a single node as input and builds a solution node-by-node. In doing so, the method returns the total tour time, travel time of the previous edge, rewards and penalties at each step, a feasibility indicator, and whether the tour is complete, i.e., if it has returned to the depot. This allows participants to consider the dynamics of the problem, considering the sampled travel times while constructing a solution.

3.3. Evaluation

The participants were evaluated over several generated instances (see Section 3.1) for each track of the competition. The competition was split into two phases, validation and test, detailed below.

Validation phase In Track 1, the teams were given a single *instance* containing $n = 55$ nodes, i.e., a single problem containing a set of nodes and other instance information (see Section 3.1). During this phase, participants were allowed to test their methods on this instance without any cap on the number of evaluations. At the end of the validation phase, the performance of each team was evaluated on 10,000 Monte Carlo samples from this instance, sampling different travel times in each new

sample. Performance was measured considering the sum of prizes and penalties of the proposed tour for each sample averaged over the 10,000 experiments. Participants were given a random seed to reproduce the same randomness in the experiments, such that the difference in results (if any) is from the performance of the algorithms.

For Track 2, participants were given 1,000 generated instances, varying in size. In total, there were 250 instances with 20, 50, 100 and 200 nodes, respectively. The performance of the proposed algorithms was evaluated on 100 Monte Carlo samples for each instance and averaged over the entire set of instances and samples, i.e., 100,000 simulations. Similarly to Track 1, the performance was measured by the sum of prizes and penalties when evaluating a single Monte Carlo sample and then averaged over all samples and instances. Note that, unlike Track 1, here we are interested in learning a policy that works well for a varying number of instances in the validation set. Participants could use the validation set to check the performance of their policies by utilizing the instances and a specific random seed for comparison to other approaches.

Test phase Only the test phase was used to determine the winners of the competition. This phase followed a similar procedure as the validation phase. In both tracks, participants had one week to submit their final test scores following the end of the validation phase. In Track 1, a new instance containing $n = 65$ nodes was generated to evaluate the final performance. Similarly, in Track 2, 1,000 instances were generated in the same fashion as in the validation phase. The procedure for evaluating performance remained unchanged from the validation phase. The instance-generating distribution was the same between the validation and test phases in both cases. Thus, an algorithm trained for the validation phase could be used to propose solutions for the instances in the test phase.

3.3.1. Submissions

In both tracks, the participants were asked to submit their output files containing the result of their proposed method as well as their implementation code for inspection by the organizing team. In Track 1, the submission file consisted of a single tour. This tour was used to compute the average performance over 10,000 Monte Carlo samples. In Track 2, participants were supposed to submit a single file containing the tours for each instance (1,000) and each Monte Carlo sample (100). In total, 100,000 tours should have been submitted for evaluating performance. This file was then used to compute the overall performance of the submission. In both cases, participants had to submit tours with size $n + 1$, where a visit to the depot determined the end of the tour.

3.3.2. Ranking submissions

Ranking submissions proceeded as follows. First, we computed the total prize and penalties for each evaluated tour. That is, for a given Monte Carlo sample j and instance I, a tour score $\alpha(\mathbf{s}^{(j)}, I)$ is computed as

$$\alpha(\mathbf{s}^{(j)}, I) = \sum_{i=1}^{n} \mathbb{1}\left[s_0^{(j)} \notin \{s_1^{(j)}, \dots, s_i^{(j)}\}\right](p_{s_i^{(j)}} + e_{s_i^{(j)}}), \tag{4}$$

where $n = 65$ in Track 1, $n \in \{20, 50, 100, 200\}$ in Track 2, and $\mathbb{1}[\cdot]$ is an indicator function that takes the value 1 when the predicate is true and 0 otherwise. The latter makes sure to stop calculating rewards when the tour returns to the starting depot. After evaluating the total number of solutions, we average the total performance obtained over all instances and Monte Carlo samples of each track. That is, the final score is given by

$$\text{score} = \frac{1}{m|\mathcal{I}|} \sum_{I \in \mathcal{I}} \sum_{j=1}^{m} \alpha(\mathbf{s}^{(j)}, I), \tag{5}$$

where $|\mathcal{I}| = 1$ and $m = 10{,}000$ in Track 1, and $|\mathcal{I}| = 1{,}000$ and $m = 100$ in Track 2. Based on this scoring metric, participating teams were ranked in descending order. That is, the teams with higher performance scores were ranked higher.

4. Winning methods

This section presents the methods that were used by the winning teams in both tracks of the competition. Though the organizers had access to the code submitted by the participants, the methods presented in this section are explained by the participants.

4.1. Track 1 winners

As there was a three-way tie in Track 1 (see Section 5), three methods are presented for this track. Participants had to make use of surrogate-based optimization techniques to optimize one instance with 65 nodes.

4.1.1. The convexers

The approach we submitted, after significant experimentation, is shaped by three observations:

1. The expected performance (i.e., the true objective) can be approximated well by using a (sampling-based) approximate model for the true probability space and an exact model for an approximation of the true probability space.
2. A learning surrogate may be used to identify search regions from where a search algorithm can be restarted.
3. Within a parallel restarted search approach, slight differences in the true objective function approximation may actually help in diversifying the parallel search efforts.

Based on these observations, our approach consists of a parallel portfolio [39,40] of restarted, hyper-parameterized [41, 42] dialectic search [43] workers that are tuned using the gender-based genetic algorithm configurator (GGA) [44–46]. Our approach features several research contributions, including a new application of hyper-parameterized dialectic search, the parallelization of dialectic search, and a surrogate-based restart method.

Parallel dialectic search Dialectic search is an iterative metaheuristic search procedure. Given a starting solution, it uses a perturbation mechanism to generate another solution from the first. The space between the solutions is then searched (e.g., using path relinking). In the case of the OPSWTW, our perturbation operator is to select a subset of clients and to permute these in such a manner that, if transition times were zero, the time windows line up to form a solution without penalties. Then, to synergize the thesis and antithesis, we path relink the two solutions: Starting at the thesis permutation, we pick each repositioned client and place it in the position prescribed by the antithesis by swapping its place with the client that currently holds that position. We commit to the swap that leads to the best solution value. Then, we consider all remaining clients currently not placed in accordance with the antithesis, pick the best one to swap, and so on, until the antithesis is reached. Among all permutations thus encountered, we pick the best and make it the new thesis.

We have shown in [41] how to *hyperparameterize* dialectic search, resulting in a search procedure that learns offline how to best solve a given instance set. Hyper-parameterized dialectic search basically extends reactive search [47], proposing to control hyperparameters of the search procedure using learned models. We use logistic regressions as the number of parameters is low, and input the same features as in [41] to the models. The models are learned using an algorithm configurator, described below.

Dialectic search can be highly parallelized, with each worker performing its own independent dialectic search and sharing information about the best solution found. Each parallel worker uses (its own local selection of) 100 randomly sampled scenarios to approximate the true objective in each local search step. When a worker encounters a solution that may improve its best solution, the variable assignment is evaluated using the exact model for a simplifying approximation of the true probability space. Since all workers use the same approximation, this provides a shared ground for accepting or rejecting a solution as improving among the parallel workers.

Learning where to restart We use a semi-supervised learning approach for forecasting the quality that will be achieved when restarting the search at a new starting point. The unsupervised part of the architecture consists of an auto-encoder that is trained offline to encode and reconstruct permutations into, respectively, from, a compressed latent space. The supervised part is learned online and aims to forecast the quality of the best solution found after restarting from a given permutation. When a worker decides to restart (which is determined by a hyperparameter), a quick local search over the latent space is conducted to find a promising starting point for the next search. This latent vector is decoded, and the resulting permutation is used to restart the search.

Training the approach Our approach contains numerous parameters and hyperparameters that must be set in advance of applying the method. We first train the autoencoder parameters offline using data collected in previous iterations of the dialectic search. The loss function is a linear combination of reconstruction loss, KL divergence and mean square error between the actual quality and predicted quality (the predicted quality is a quadratic function on top of the encoder). We use an ADAM optimizer to optimize the loss function. We separate this training from the configuration of the overall approach as it would be too expensive to couple the parameters of the model with the hyperparameters of the dialectic search. We note, however, that this could be an interesting avenue for future work. The dialectic search exposes its hyperparameters (described briefly above) and these can be configured using an algorithm configurator. We use GGA [46] and configure the hyperparameterized dialectic search on a subset of the instances provided in the first phase of the competition.

Evaluated quality of the approach This approach solves the test instance in about 90 seconds using roughly 35 minutes of raw compute time over all workers. This efficiency allowed us to also solve all 1,000 instances from track 2 of the competition, which led to a test performance (over the 100 scenarios per instance prescribed in the competition) of 10.8106 (whereby the expected value in the simplified probability space is 10.78, so the scenarios used in the competition evaluation are comparably friendly). Note that this means that using the best default tours (and sticking to them, no matter how the actual transit times evolve) actually outperforms the winning reinforcement solution of Track 2.

4.1.2. Margaridinhas

The method proposed by the *Margaridinhas* for Track 1 of the AI for TSP competition has three main components: (1) a mixed-integer surrogate model, (2) an iterative approach, and (3) a genetic algorithm. The iterative approach improves the surrogate model throughout iterations according to a fixed set of parameter values and outputs the best route found at

the end. The genetic algorithm combines solutions output by distinct calls to the iterative approach (with different sets of parameter values) to find even better solutions.

(1) Surrogate model The surrogate model outputs a route that maximizes the deterministic reward plus the estimated penalty based on previous simulations within the iterative approach. Each node $i \in \mathcal{N}$ in the nodes set $\mathcal{N}$ has a retrievable reward r_i, and each arc $(i, j) \forall\ i, j \in \mathcal{N}$ has an estimated penalty p_{ij}. The decision variable $x_{ij} \in \{0, 1\}$ equals 1 if arc (i, j) is in the route or 0 otherwise. For the sake of brevity, let $\mathcal{T}$ be the set of values for $\{x_{ij}\}_{i,j \in \mathcal{N}}$ describing a route that starts and ends at the depot and does not contain cycles nor subroutes. Let also *MaxRouteSize* be a parameter controlled externally by the iterative approach that represents the maximum route size. The surrogate model $\mathcal{M}(MaxRouteSize)$ is formulated as follows.

$$\mathcal{M}(MaxRouteSize): \quad \max_{x \in \mathcal{T}} \quad \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{N}} (r_j + p_{ij}) \cdot x_{ij} \tag{6a}$$

$$\text{subject to:} \quad \sum_{i \in \mathcal{N}} \sum_{j \in \mathcal{N}} x_{ij} \leq MaxRouteSize \tag{6b}$$

The objective function (6a) maximizes the deterministic reward of the route plus the estimated penalty. Constraint (6b) bounds the number of arcs in the route to the maximum route size parameter *MaxRouteSize*. Note that set $\mathcal{T}$ does not enforce the maximum duration T nor the time windows at nodes. The surrogate model $\mathcal{M}(MaxRouteSize)$ learns what arcs to avoid when building a route through better estimated penalties p_{ij} in the objective function, and added cuts according to previously visited solutions.[3]

(2) Iterative approach The iterative approach explores the solution space of the problem according to the parameters, finding distinct solutions accordingly. The iterative approach has four parameters: the maximum number of iterations K, the number of simulations per iteration M, the feasibility threshold *FeasibilityThreshold*, and the gap threshold *GapThreshold*.

First, the iterative approach cuts nodes and arcs from the surrogate model $\mathcal{M}(MaxRouteSize)$ that would never be visited in a feasible route (e.g., nodes with a time window starting after the maximum duration T), and sets *MaxRouteSize* to 2. At each iteration, the approach solves the surrogate model $\mathcal{M}(MaxRouteSize)$ to obtain a solution x^k and simulates the associated route s^k. If route s^k is feasible in at least *FeasibilityThreshold* percent of the M simulations, the approach stores it and cuts the feasible solution x^k from the feasible region. Otherwise, the approach cuts the infeasible structure given by the infeasible solution x^k (i.e., a sequence of arcs, excluding the return to the depot, that is most likely never feasible).

If the gap between the best solution found so far and the upper bound for routes with size limited by *MaxRouteSize* is less than *GapThreshold*, the approach increases *MaxRouteSize* by 1 and starts the search for larger routes. The upper bound for routes with size limited by *MaxRouteSize* can be calculated by solving the surrogate model $\mathcal{M}(MaxRouteSize)$ without penalties p_{ij}. Next, the approach updates the penalties p_{ij} in the objective function according to the penalty of route s^k given by the M simulations. The total penalty of route s^k is equally divided between its arcs, and the penalty p_{ij} of an arc (i, j) is simply the average over registered penalties. The algorithm stops and outputs the best route once the best solution meets the upper bound or the approach hits the maximum number of iterations K.

(3) Genetic algorithm A genetic algorithm (GA) is used as a local search policy to improve the set of solutions found through distinct calls to the iterative approach. We run the GA for a few generations on 14 warmed-up solutions output by the iterative approach, with scores ranging from 9.83 to 11.28, to investigate the neighborhood. The iterative approach takes on average 15 minutes to find a warmed-up solution on a Windows computer with an Intel Core i7-10510U CPU @ 1.80 GHz × 8 processor and 8 GB RAM memory. The initialization step of the GA uses the 14 warmed-up solutions together with random solutions to make 25600 in total. Even though there were only a few surrogate solutions, they were crucial for the GA to output better solutions rapidly. We ran 5 generations of the GA, performing 100 evaluations for each new solution. After the evaluation of a generation, we reduce the number of solutions to 200 parents by picking the winners of a tournament and the top solutions from past generations. We use the Non-Wrapping Ordered Crossover reproduction operator from [48] since it preserves the order of the nodes.

The GA has managed to successfully find better solutions than the distinct calls to the iterative approach. We found a number of solutions scoring 11.32, which is the optimal solution for the Track 1 test instance, after running the GA for 2 hours in the previously described setup. Fig. 2 presents the estimated probability that a node occurs in a certain position of the optimal solution. These estimates are based on the optimal solutions found through the GA. Note that in Fig. 2 the optimal solutions share the exact same set of nodes; moreover, the order of visit of some neighboring nodes can be swapped. Further details about the method and how it has been implemented can be found on GitHub.[4]

[3] In alternative to the surrogate model, it could be chosen a random or constructive initialization method. The former showed to perform poorly, while the latter was competitively good compared to the surrogate model. The advantages of using the surrogate model lie in the fact that the overall pipeline with the surrogate model requires fewer iterations to find the optimal solution, it performs better for smaller populations and it seems more robust to stochasticity.

[4] https://github.com/almeidawarley/tsp-competition.

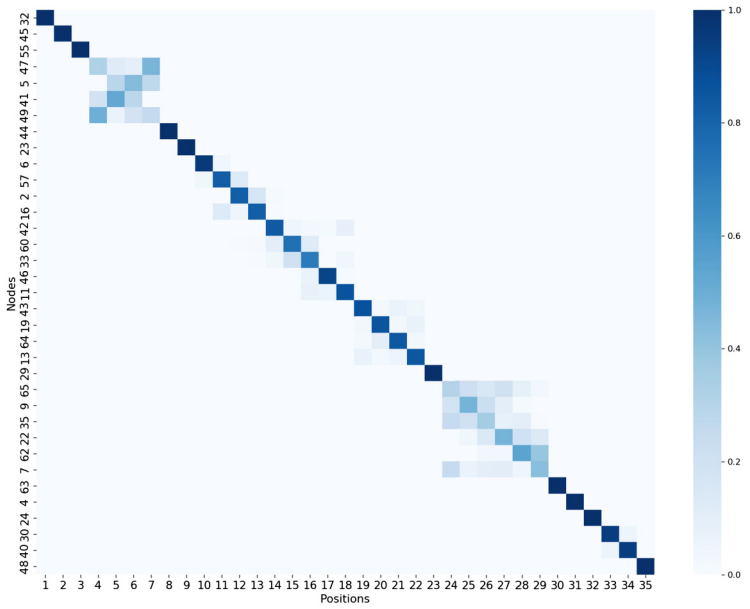


Fig. 2. Estimation of node probability of occurrence in the positions of optimal solutions. The first visited node is always 32. The nodes visited in 4th to 7th places are {47, 5, 41, 49}.

4.1.3. ZLI

The approach of the ZLI Team to the competition is based on the assumption that the problem is a black box and the algorithm only has access to the number of nodes n and the values returned by the objective function f. By controlling the number of evaluations per solution used to estimate its expected objective value (Eq. (6a)), that is, the fidelity of the estimation, it is possible to evaluate hundreds of thousands of solutions within the time limit of the competition. Therefore, from the point of view of classical Bayesian optimization, the problem cannot be considered expensive, and classical approaches based on Gaussian Process regression (GPR), such as CEGO [49], are not feasible due to the large number of evaluation points. However, a good estimation still requires thousands of evaluations and, thus, it is important to keep track of each estimated value and its level of fidelity to progressively increase it while avoiding redundant evaluations. Thus, we apply multiple techniques simultaneously to address these limitations: (1) reducing the dimensionality of the problem by excluding infeasible nodes, (2) caching the expected objective value of evaluated solutions for various fidelity levels, (3) training a classifier to predict the probability of feasibility of a solution, (4) applying a self-adaptive black-box evolutionary algorithm (EA) directly to the objective function while increasing the fidelity, (5) supporting the EA by a surrogate model based on GPR, and (6) enabling the model to deal with many evaluation points via a clustering approach.

As a first step, we reduce the dimensionality of the problem by evaluating all possible tours that visit exactly one node 1 000 times. If the result is infeasible in at least 10% of the repeated evaluations, the respective node is removed from the set of available nodes. In the competition instance, this step reduced the dimension from $n = 65$ to $n = 37$.

Afterward, we step-wise increase the fidelity of the objective function, i.e., increasing the number of repeated evaluations for each candidate tour, in order to find a set of reasonably good solutions. For each of fidelity level of 1, 10 and 100, we run a self-adaptive EA with a budget of 10 000 evaluations. The first EA run starts from random tours, and each subsequent run starts from the last population of the preceding (lower-fidelity) run.

The EA roughly follows the Mixed Integer Evolution Strategy [50]. Its self-adaption dynamically configures the variation (recombination and mutation) operator choices, and the mutation rate. The variation operators are mutated randomly with probability p and are recombined by choosing randomly from parents. The mutation rate is itself mutated by: $q^* = q \exp(\tau z)$ with learning rate τ and z being a sample from the normal distribution (zero mean, unit variance). The mutation rate is recombined via intermediate crossover. The possible choices for variation operators are standard operators for permutations (recombination: Cycle, Order, Position, Alternating Position; mutation: Swap, Insert). We modified these mutation operators to avoid creating solutions that visit less than two nodes (since we already tested all one-node tours during dimensionality reduction) and to avoid mutations that occur in the inactive part of a tour (changing only unvisited nodes).

The EA avoids re-evaluating solutions with the same level of fidelity by maintaining a cache memory of expected values per solution. This cache memory is shared between EA runs. The cache memory after the three initial EA runs is used to train a Gradient Boosting Decision Tree Classifier to predict the probability of a tour being infeasible based on the rank representation of the tour.

The EA is run a fourth time with a fidelity of 100 and a budget of 10 000 using the trained classifier to avoid evaluating solutions if their predicted probability of being feasible is less than 0.4.

A fifth run of the EA with a fidelity of 100 and a budget of 5 000 incorporates a surrogate model, a Gaussian process regression model (GPR, aka Kriging). At its core, GPR is based on a kernel function that captures the similarity or correlation between input points, i.e., tours s and s'. To enable the GPR model to deal with the discrete sequence points, we use an exponential kernel $k(x, x') = \exp(-\theta d_{\mathrm{L}}(x, x'))$, where $d_{\mathrm{L}}(x, x')$ is the Levenshtein distance between two tours. This distance measure counts substitutions, deletions, and insertions of nodes required to turn one tour into another. Only the active part of the tour is considered during distance calculation, i.e., nodes that are actually visited.

Since the surrogate-assisted EA will evaluate thousands of solutions, we need to be able to model large numbers of points. This is usually a challenge for GPR models, as the required computational effort increases significantly. To deal with this issue, we separate the training data into subsets via clustering, train a GPR model for each cluster, and combine the individual models into an ensemble via a weighted sum. The ensemble prediction uses the predicted uncertainty of each individual model for weighting. This approach of clustered Kriging is described by Wang et al. [51]. To further save computational effort, the model is only retrained with new data every 20 generations. In each EA generation, the surrogate model is used to pre-filter the generated offspring, removing the worst 50%.

We run the EA two additional times. The sixth run simply increases the fidelity to 500. The seventh and final run of the EA uses the probability predicted by the classifier as an additional criterion, together with the prediction of the GPR model, to sort the generated offspring instead of simply discarding solutions that are predicted to be infeasible.

Since each run of the EA starts from the last population of the previous run, the seven EA runs can be seen as different phases of a single overall run of the algorithm, with the shared cache memory acting as a global archive of the best solutions found that is also used to re-train the classifier between runs.

In the last step, the best 250 solutions from the cache memory, which contain the best from the last run of the EA, are evaluated with a fidelity of 10 000 evaluations and the best one is returned.

Experiments were carried out on a cluster of 2 × 8-core Intel Xeon E5-2650v2 2.60 GHz with 64 GB RAM. We launched multiple independent runs of the above algorithm in parallel to assess the robustness of the results, however, each run is executed sequentially on a single core. A complete run of the above algorithm requires an average of 12 hours to complete. The code is made available on Zenodo.[5]

4.2. Track 2 winners

In this track, participants had to use deep reinforcement learning methods to find optimal policies for 1,000 instances with sizes ranging from 20 to 200 nodes.

4.2.1. RISE up

Our solution approach for the challenge consists of three components. First, we use the POMO reinforcement learning approach proposed by Kwon et al. [9] to learn one policy per problem size. Next, we use efficient active search [52] to fine-tune the learned policies for each instance being solved, thus creating an individualized policy for each instance of the test set. Finally, we use Monte-Carlo rollouts to construct the final solutions. In the following, we will describe each of these three components in more detail. The code for our method and the trained models are available online.[6]

POMO We use POMO to learn an initial policy for each problem instance size. The POMO approach is an end-to-end deep reinforcement learning approach for combinatorial optimization problems. POMO is based on the REINFORCE algorithm but exploits symmetries in combinatorial optimization problems to encourage exploration during learning. The network architecture of the employed model is based on the transformer architecture and consists of an encoder and a decoder neural network. The implementation of POMO is made available by the authors. POMO can be used to solve multiple instances in a batch, as well as to perform multiple rollouts per instance in parallel.

We slightly adjust the POMO implementation to support the OPSWTW. We change the input that POMO expects for each node i to $(x_i, y_i, p_i, l_i, h_i)$. All values are scaled before being input into POMO's deep neural network. We scale the x_i and y_i-coordinates based on the 2D-space limits, p_i based on the maximum prize per instance, and l_i and h_i based on the given maximum tour time T. Furthermore, we change the decoder context to include the current time $L(\mathbf{s})$ (also scaled by T), the embedding of the current node, and the embedding of the depot. Finally, we adjust the masking schema of POMO to forbid actions that correspond to traveling to a node i where $l_i > T$ or where $h_i < L(\mathbf{s})$ as well as previously visited nodes.

We train one separate policy model for each of the four considered problem sizes. The training set has been generated using the provided instance generator. Each model is trained for several days until full convergence on a single Tesla V100 GPU. For the larger instances with $n = 100$ and $n = 200$ we use transfer learning and start the training from the policy model trained on $n = 50$.

Efficient active search Efficient active search (EAS) [52] is a method that uses reinforcement learning to fine-tune a policy to a single test instance. In contrast to the original active search [4], it only adjusts a small subset of (model) parameters.

[5] doi: 10.5281/zenodo.7015507.

[6] https://github.com/FynnSu/AI_for_TSP.

This allows us to solve multiple instances in parallel, resulting in significantly reduced runtime. For each test instance, we create a copy of the corresponding problem-size specific policy learned by POMO and then fine-tune a subset of the policy parameters to that specific instance. During this process we generate 18 million solutions per instance, which takes up to 30 minutes on our GPU for the larger instances with 200 nodes. The result is a set of 1,000 separate policies. Note that the travel times between nodes are not fixed during the training process. Instead, the travel times are sampled for each solution construction process. This enables EAS to learn a robust policy that can create high-quality solutions for a wide range of scenarios. If we use the fine-tuned policies to construct solutions greedily, we observe an average reward of 10.67 on the test set. In contrast, using the four size-specific models for solution construction results in a reward of 10.43.

Monte-Carlo rollouts We construct the solutions for the test instances using Monte-Carlo rollouts and the instance-specific policies learned via EAS. We construct a solution for a test instance as follows. At each decision step, we first use the policy model to generate a probability distribution over all possible actions (i.e., all possible nodes that can be visited next). For the five actions with the highest associated probability values, we then perform 600 Monte-Carlo rollouts each. Each Monte-Carlo rollout starts with the corresponding actions and then completes the solution by sampling the following actions according to the learned policy model. Once all Monte-Carlo rollouts are finished, the action with the highest average reward is selected. Note that only after the final action has been chosen are the actual travel times between the nodes revealed. For the Monte-Carlo rollouts, travel times are sampled independently in the preceding step. Using Monte-Carlo rollouts increases the average reward to 10.77 (from 10.67 obtained via a greedy solution using the EAS-based policies).

4.2.2. Ratel

The approach of team *Ratel* relies on a version of the Pointer Networks (PN) designed to tackle problems with dynamic time-dependent constraints, in particular, the Orienteering Problem with Time Windows [13]. While the model shares the same basic structure with previous PNs [4,5], i.e. a set encoding block that encodes each node, a sequence encoding block that encodes the constructed sequence so far and a pointing mechanism block, its architecture differs from previous PNs mainly in that it introduces recurrence in the node encoding step. This recurrence makes it so that both encoding and decoding steps are carried out sequentially for every step of the solution construction process. This aspect brings great advantages, especially when solving problems with dynamic constraints, as it allows the use of masked self-attention using a lookahead-induced graph structure, which in turn allows for an updated representation of each admissible node in every step [13].

Input features Since the model is recursive, at each step it determines a feature vector associated with every admissible node. These feature vectors are given as input to the set encoder in order to compute a step-dependent representation of each node. This feature vector is a combination of static features – that remain constant throughout the solution construction process – and dynamic features – that can change at every step. For this problem, the model uses 11 static features and 34 dynamic features.

The static features are obtained directly from each instance of data. Concretely, the Euclidean coordinates of each node i (x_i and y_i), opening and closing time (l_i and h_i), time window width ($h_i - l_i$), prize (p_i), maximum time available (i.e. max length, T), prize divided by time window width, prize divided by distance from the depot, and the difference between maximum time available and opening and closing times ($T - l_i$ and $T - h_i$). As for dynamic features, the model uses 34 features that are functions of the current time and current node. Some of the dynamic features are boolean and indicate whether a node's feasibility conditions are satisfied, assuming either the fastest travel times possible or the worst travel times. Some examples of non-boolean dynamic features are, for each node, the time left until the opening time, the time left until closing time, the fraction of time elapsed since tour start, prize divided by max time to arrive at the node, prize divided by time to closing time, the probability of arriving after closing time, the probability of arriving after the maximum time available and the expected prize of the node.

Setup and training In order to speed up training and model development, training was done by sampling from 804,000 pre-generated instances: 4,000 instances for each number of nodes between 10 and 210 nodes, with seeds ranging from 1 to 4000. During training, at each step, the model samples one instance from the pre-generated set of instances (one number of nodes between 10 and 210 and one seed between 1 and 4,000) with replacement. Note that the seeds from the validation phase (12345) and the test phase (19120623) are outside this range and thus were not considered during training. This is to avoid/minimize leakage and overfitting and achieve a more realistic final collected prizes.

The model was trained for 15,000 epochs, using a method based on the REINFORCE algorithm [53] as in [13], with entropy regularization to enhance exploration, L_2 regularization and Adam optimizer. In each epoch, 6 simulations on the same instance were performed, each one composed of 32 sample solutions, equaling a batch size of 192 sampled solutions. Further implementation details and hyperparameter values can be found in GitHub.[7]

[7] https://github.com/mustelideos/td-opswtw-competition-rl.

Inference During evaluation, solutions are constructed using a greedy strategy, i.e. at each time step, we select the node for which the model gives the highest probability. All experiments were performed on a 6 core CPU at 1.7 GHz (Intel Xeon CPU E5-2603 v4) with 12 GB of RAM and an Nvidia GeForce GTX 1080 Ti GPU. The final model took up to 48 hours to train and around 12 hours to generate the validation/test submission files.

5. Results and discussion

This section presents the results of all participating teams in the validation phase and final test phase. In addition, we implemented a heuristic as a benchmark method to compare to the performance of the winning methods.

5.1. Benchmark heuristic

We have implemented a classical heuristic baseline for both tracks that does not make use of machine learning, namely Adaptive Large Neighborhood Search (ALNS) [54]. ALNS has shown its good performance in solving orienteering problems in the literature (i.e., [55,56]). This heuristic is based on the ruin-and-create principle of Large Neighborhood Search (LNS), where solutions are gradually improved via continuous destroy and repair operators. ALNS extends this by allowing multiple destroy and repair operators to be used within the search. For this, the destroy and repair operators are assigned a weight to determine which operators are selected in each iteration of the algorithmic search. These weights are dynamically adjusted based on operator performances; more successful operators are given higher weights and are, therefore, more likely to be selected in a next iteration of the search.

We defined destroy operators based on *random remove* and *random edge remove*. In random remove, n customers are removed uniformly for a solution, and in random edge remove, a sequence of customers is removed. For both destroy operators, we defined two alternatives: a 'modest' and a 'severe' variant, removing between 0 - 25% and 20 - 40% of the customers from a solution. Three repair operators are defined to reinsert customers in the destroyed solution: *random distance repair*, *random price repair*, and *random ratio repair*. In random distance repair, we randomly select a number of customers and insert them into the solution at their least expensive position (in terms of distance). Random prize repair and random ratio repair work according to the same principle, yet insert nodes sequentially at the position where the total accumulated rewards are maximized, or the ratio between reward and additional distance traveled is optimized. Simulated Annealing is selected as the acceptance criterion for the search with a starting temperature of 1 and a linear decay step of 0.01 for cooling, bounded at 0.25. We use the original scoring function of ALNS and update weights using convex combinations of the current weights and the score obtained for the current candidate solution. Search is performed for 100 iterations, regardless of the instance size. This baseline was implemented after the competition and was therefore not available to participants. The code for the implemented ALNS is available online.

For track 1, if participants manage to develop an SBO approach tailored to the problem at hand, e.g. by dealing with the constrained combinatorial search space, the expectation is that it can outperform classical heuristic approaches that do not make use of machine learning, such as ALNS. The reason for this is that a surrogate model deals with two aspects of the problem at the same time: 1) the expensive objective, and 2) the stochasticity. By replacing the objective with a deterministic and cheap to compute surrogate, these aspects of the problem are circumvented. Such a method would still need to evaluate the expensive objective but is better suited for a low number of these evaluations than traditional heuristic approaches [34].

For track 2, we expect new methods based on DRL to be able to outperform the ALNS for this problem. The reason for this is that several works have proven that DRL can find better heuristics through learning [14,15], while classical heuristics make decisions based on hand-crafted rules. If enough data and computation time for learning is available, these learning capabilities could be leveraged to improve upon hand-crafted heuristics.

5.2. Results

Table 2 shows the score at the deadline of each phase. Only the test phase was relevant for deciding on the winners of the competition. As can be seen, nine teams participated in the validation phase and seven in the test phase (disregarding the benchmark heuristic ALNS). The leaderboard with results was visible to all participants and was updated every time a participating team submitted a solution. During the validation phase, we saw a large improvement in the scores, indicating that teams were actively improving their methods. It can also be seen that at the end of the test phase, there was a three-way tie for the top participants of Track 1. Although this was not the case from the beginning of this phase, during the test phase, these three participants managed to find the globally optimal solution for the instance under consideration. The prize money for first and second place in Track 1 was split among these three teams (ZLI, Margaridinhas, and Convexers), while the prize money for first and second place in Track 2 was given to teams RISEup and Ratel, respectively.

Comparison with ALNS To further analyze the performance of the learning-based solutions, we have compared the performances of the winning solutions of both tracks with the ALNS baseline solution. For Track 1, it can be seen in Table 2 that ALNS did not find the globally optimal solution or come close to the results of the top three teams, though it did outperform one of the non-winning teams. This indicates that it can be beneficial to apply learning-based solutions to difficult routing

Table 2
Leaderboard for the validation phase and the final test phase at the deadline. The results of ALNS were added after the competition.

Track 1 (SBO)	Validation phase	Track 1 (SBO)	Test phase
ZLI	8.404499999999793	Convexers	11.320000000002786
Convexers	7.8099999999988885	Margaridinhas	11.320000000002786
Margaridinhas	7.533834999999982	ZLI	11.320000000002786
Topline	3.540000000000663	ALNS	6.91096999999927
VK	0.5300000000000102	Topline	4.300000000000301
Track 2 (DRL)	**Validation phase**	**Track 2 (DRL)**	**Test phase**
Ratel	10.69104	RISEup	10.77341
ML for TSP	9.82955	Ratel	10.58859
RISEup	9.16567	ML for TSP	10.39341
VK	-6.69772	ALNS	4.94231
UniBw	-13.14861	-	-

Table 3
Validation comparison per instance sizes.

solution method	mean	std	min	25%	50%	75%	max
	Instances 0-250 (20 nodes)						
RISEup	5.41	0.28	3.51	5.44	5.46	5.46	5.47
Ratel	5.32	0.79	1.19	5.48	5.50	5.51	5.53
ALNS	4.75	1.26	1.48	4.30	5.25	5.53	5.74
	Instances 251-500 (50 nodes)						
RISEup	8.22	0.19	6.71	8.22	8.24	8.26	8.27
Ratel	8.08	1.01	0.67	8.19	8.23	8.25	8.31
ALNS	4.88	2.78	0.08	2.70	5.89	6.88	8.31
	Instances 501-750 (100 nodes)						
RISEup	11.62	0.10	10.99	11.60	11.64	11.66	11.67
Ratel	11.39	1.16	2.71	11.45	11.55	11.64	11.71
ALNS	4.79	3.33	0.00	1.44	5.74	7.30	9.82
	Instances 751-1000 (200 nodes)						
RISEup	17.87	0.45	14.30	17.87	17.93	17.97	18.03
Ratel	17.57	1.22	7.37	17.54	17.72	17.84	18.07
ALNS	5.36	3.86	0.00	1.26	6.27	8.30	11.74

problems. For Track 2, we solved the sets of instances with different sizes (250 instances with 20, 50, 100 and 200 nodes respectively) and compared them with solutions from RISEup and Ratel. The performances of the three methods are shown in Table 3, comparing the mean, min, max, standard deviations, and quartiles of 50 runs of ALNS for 100 iterations of search with the solutions of the top two competition participants. It can be observed that the learning-based approach performs better in solving instances of the largest size, whereas the best found solutions of the ALNS algorithm perform better on the smallest instances. This again indicates that it can be beneficial to apply learning-based solutions to difficult routing problems, but also that using classical heuristics might be sufficient for easier problems.

To further investigate our claims, we selected a difficult to solve solution from the largest set of instances, 'instance0853', and explored how the ALNS baseline performs when it is provided with additional search budget. For this, we increased the number of iterations from 100 to 250. The performances are compared with the RISEup winning approach of track 2 and are displayed in Fig. 3. These results indicate that, for complicated routing problems as in this competition, a learning-based approach can have a more positive impact than using a much larger search budget with a classical approach.

5.3. Competition insights

We have obtained several insights from organizing this competition. First of all, our main goal of advancing the state-of-the-art of ML methods in routing problems has been achieved, with seven new surrogate or reinforcement learning methods presented in this work by the winning participants, and other methods having been applied by the other participants. Our other goals, namely attracting ML researchers to solve a challenging routing problem, and creating benchmark problem instances for ML-based approaches, have also been achieved, making this a successful competition in the eyes of the organizers.

What we noticed was that having clear restrictions on what is and what is not allowed in the competition stimulated creativity. By limiting participants to use SBO and DRL approaches, new interesting methods have been developed. Another aspect of the competition that was beneficial for everyone was a chat function, where any confusion was quickly cleared up.

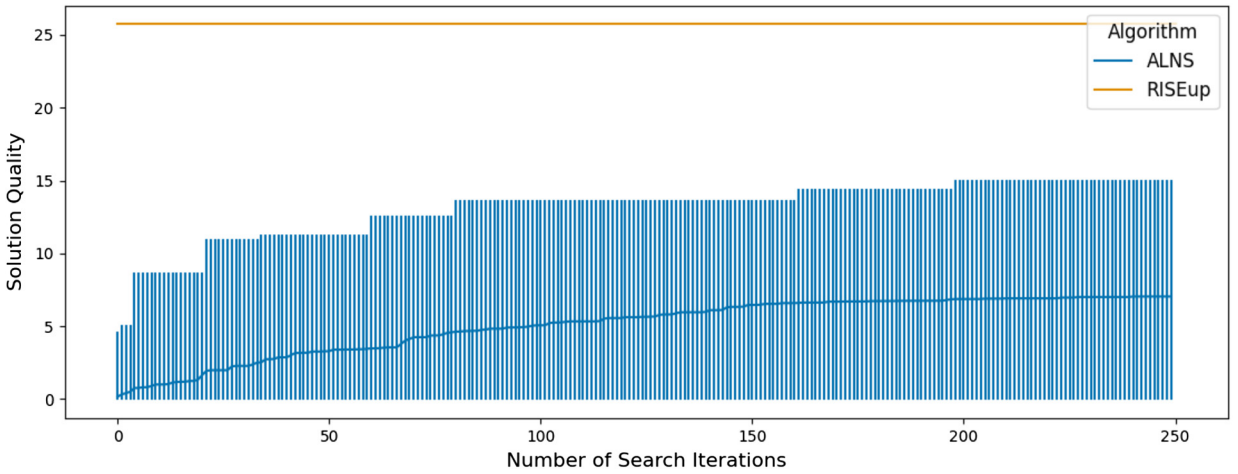


Fig. 3. ALNS solution quality over time for 'instance0853'.

There are also points of improvement for future (similar) competitions. It remains questionable whether providing the whole simulator to participants is really the way to go in an ML competition. Although the samples of random variables in the simulator were not known in advance, the exact probability distributions could be retrieved by looking at the code to which participants had access. This was known by the organizers, but it was decided to not obfuscate this information so that it remained easy for participants to run the provided simulator and use it for ML.

Another point of improvement is the difficulty of the problem, especially for Track 1. Because we did not want to slow down the solution evaluation process on our server too much, and because we noticed solutions in the validation phase were still being improved, we decided to keep the instance size somewhat limited. Over the course of the test phase, the methods of the participants were improved up to the point that three of the methods managed to find the globally optimal solution, causing a three-way tie. Although this shows the power of these three different methods, for the purpose of the competition, it is best if the organizers do not underestimate the participants and instead are inclined towards too difficult instances, even though our instance proved too difficult for a classical heuristic. This should hopefully push the creativity of participants and the power of their methods even further.

Furthermore, the organizers did not provide any working baseline methods, and it was recognized that the entry level to the competition was rather high. Therefore, most participants are those who have good knowledge of both optimization algorithms and machine learning techniques. To promote the line of research (AI and ML for solving optimization problems) and to attract more participants, it would be very beneficial if baseline methods are provided at the beginning of the competition.

For future work, we will consider the points of improvement, and in addition, see whether the winning methods can be generalized to be used in other routing problems.

6. Conclusion

We reported on the first international competition on AI4TSP. The participants were asked to solve an orienteering problem with stochastic weights and time windows (OPSWTW). We described the setup of two tracks, focusing on two learning techniques from AI: surrogate-based optimization and deep reinforcement learning. We described the approaches of the winning teams of two tracks and gave an overview of the results. Furthermore, we explained the developed simulation model, which was used to generate the training and testing instances of OPSWTW, and was coupled with two implemented baseline algorithms of two tracks. In addition, the simulator used in this competition and the code of some of the winning approaches of both tracks of the competition has been made publicly available. The simulation model with various algorithms can serve as a benchmark for researchers to develop and compare surrogate-based and reinforcement learning-based approaches to stochastic routing problems.

This paper was written together by the organizers and the winning teams of the competition. The organizers were pleased with the outcome, as the purposes of the first AI4TSP competition have been achieved. The results show the diversity of the adopted methods, from pure machine learning approaches to integrating learning with more traditional heuristics. There is great potential for developing other sophisticated algorithms, especially in leveraging machine learning for expert-crafted heuristics, to solve (stochastic) routing problems.

Looking forward, the organization team will continue AI4TSP by considering various improvements such as setting time and computation budget and implementing more realistic distribution functions in the simulator. In addition, other practical and societally relevant optimization problems will be considered in future editions of the competition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared the link to data/code in the paper.

Acknowledgements

The organizers would like to thank Ortec and Vanderlande for sponsoring the prize money.

References

[1] Y. Bengio, A. Lodi, A. Prouvost, Machine learning for combinatorial optimization: a methodological tour d'horizon, Eur. J. Oper. Res. 290 (2021) 405–421.
[2] O. Vinyals, M. Fortunato, N. Jaitly, Pointer networks, in: Proceedings of the 29th Conference on Neural Information Processing Systems (NIPS), 2015, pp. 2692–2700.
[3] C.K. Joshi, T. Laurent, X. Bresson, An efficient graph convolutional network technique for the travelling salesman problem, arXiv:1906.01227, 2019.
[4] I. Bello, H. Pham, Q.V. Le, M. Norouzi, S. Bengio, Neural combinatorial optimization with reinforcement learning, arXiv:abs/1611.09940, 2017.
[5] W. Kool, H. Van Hoof, M. Welling, Attention, learn to solve routing problems!, arXiv preprint, arXiv:1803.08475, 2018.
[6] Q. Ma, S. Ge, D. He, D. Thaker, I. Drori, Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning, arXiv preprint, arXiv:1911.04936, 2019.
[7] M. Nazari, A. Oroojlooy, L.V. Snyder, M. Takác, Reinforcement learning for solving the vehicle routing problem, in: NeurIPS, 2018.
[8] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, L.-M. Rousseau, Learning heuristics for the TSP by policy gradient, in: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer, 2018, pp. 170–181.
[9] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, S. Min, POMO: policy optimization with multiple optima for reinforcement learning, in: H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, vol. 33, Curran Associates, Inc., 2020, pp. 21188–21198.
[10] Z.-H. Fu, K.-B. Qiu, H. Zha, Generalize a small pre-trained model to arbitrarily large tsp instances, arXiv preprint, arXiv:2012.10658, 2020.
[11] W. Kool, H. van Hoof, J. Gromicho, M. Welling, Deep policy dynamic programming for vehicle routing problems, arXiv preprint, arXiv:2102.11756, 2021.
[12] A. Delarue, R. Anderson, C. Tjandraatmadja, Reinforcement learning with combinatorial actions: an application to vehicle routing, Adv. Neural Inf. Process. Syst. 33 (2020).
[13] R. Gama, H.L. Fernandes, A reinforcement learning approach to the orienteering problem with time windows, Comput. Oper. Res. 133 (2021) 105357.
[14] A. Hottung, K. Tierney, Neural large neighborhood search for the capacitated vehicle routing problem, in: ECAI 2020, IOS Press, 2020, pp. 443–450.
[15] P.R.d.O. da Costa, J. Rhuggenaath, Y. Zhang, A. Akcay, Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning, in: S.J. Pan, M. Sugiyama (Eds.), Proceedings of the 12th Asian Conference on Machine Learning, Proceedings of Machine Learning Research, PMLR, vol. 129, 2020, pp. 465–480.
[16] P. da Costa, J. Rhuggenaath, Y. Zhang, A. Akcay, U. Kaymak, Learning 2-opt heuristics for routing problems via deep reinforcement learning, SN Comput. Sci. 2 (2021) 1–16.
[17] Y. Wu, W. Song, Z. Cao, J. Zhang, A. Lim, Learning improvement heuristics for solving routing problems, arXiv preprint, arXiv:1912.05784, 2019.
[18] H. Lu, X. Zhang, S. Yang, A learning-based iterative method for solving vehicle routing problems, in: International Conference on Learning Representations, 2020.
[19] X. Chen, Y. Tian, Learning to perform local rewriting for combinatorial optimization, in: Advances in Neural Information Processing Systems, vol. 32, 2019, pp. 6281–6292.
[20] L. Xin, W. Song, Z. Cao, J. Zhang, NeuroLKH: combining deep learning model with Lin-Kernighan-Helsgaun heuristic for solving the traveling salesman problem, in: Advances in Neural Information Processing Systems, 2021.
[21] S. Li, Z. Yan, C. Wu, Learning to delegate for large-scale vehicle routing, in: Advances in Neural Information Processing Systems, 2021.
[22] L. Gao, M. Chen, Q. Chen, G. Luo, N. Zhu, Z. Liu, Learn to design the heuristics for vehicle routing problem, arXiv preprint, arXiv:2002.08539, 2020.
[23] R. Zhang, A. Prokhorchuk, J. Dauwels, Deep reinforcement learning for traveling salesman problem with time windows and rejections, in: 2020 International Joint Conference on Neural Networks (IJCNN), 2020, pp. 1–8.
[24] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz, A.A. Cire, Combining reinforcement learning and constraint programming for combinatorial optimization, Proc. AAAI Conf. Artif. Intell. 35 (2021) 3677–3687.
[25] J. Sui, S. Ding, R. Liu, L. Xu, D. Bu, Learning 3-opt heuristics for traveling salesman problem via deep reinforcement learning, in: Asian Conference on Machine Learning, PMLR, 2021, pp. 1301–1316.
[26] M. Kim, J. Park, J. Kim, Learning collaborative policies to solve NP-hard routing problems, in: Advances in Neural Information Processing Systems, 2021.
[27] A. Hottung, B. Bhandari, K. Tierney, Learning a latent search space for routing problems using variational autoencoders, in: International Conference on Learning Representations, 2021, https://openreview.net/forum?id=90JprVrJBO.
[28] L. Bliek, S. Verwer, M. de Weerdt, Black-box combinatorial optimization using models with integer-valued minima, Ann. Math. Artif. Intell. (2020) 1–15.
[29] R. Karlsson, L. Bliek, S. Verwer, M. de Weerdt, Continuous surrogate-based optimization algorithms are well-suited for expensive discrete problems, in: Proceedings of the Benelux Conference on Artificial Intelligence, 2020, pp. 88–102.
[30] M. Namazi, C. Sanderson, M.A.H. Newton, A. Sattar, Surrogate assisted optimisation for travelling thief problems, in: SOCS, 2020, pp. 111–115.
[31] M.-Y. Fang, J. Li, Surrogate-assisted genetic algorithms for the travelling salesman problem and vehicle routing problem, in: 2020 IEEE Congress on Evolutionary Computation (CEC), 2020, pp. 1–7.
[32] A. Bracher, N. Frohner, G.R. Raidl, Learning surrogate functions for the short-horizon planning in same-day delivery problems, in: CPAIOR, 2021.
[33] M.A. Ardeh, Y. Mei, M. Zhang, A GPHH with surrogate-assisted knowledge transfer for uncertain capacitated arc routing problem, in: 2020 IEEE Symposium Series on Computational Intelligence (SSCI), 2020, pp. 2786–2793.
[34] B. Shahriari, K. Swersky, Z. Wang, R. Adams, N.D. Freitas, Taking the human out of the loop: a review of Bayesian optimization, Proc. IEEE 104 (2016) 148–175.
[35] C. Verbeeck, P. Vansteenwegen, E.-H. Aghezzaf, Solving the stochastic time-dependent orienteering problem with time windows, Eur. J. Oper. Res. 255 (2016) 699–718.

[36] Y. Dumas, J. Desrosiers, E. Gelinas, M.M. Solomon, An optimal algorithm for the traveling salesman problem with time windows, Oper. Res. 43 (1995) 367–371.
[37] M. Fischetti, J.J.S. Gonzalez, P. Toth, Solving the orienteering problem through branch-and-cut, INFORMS J. Comput. 10 (1998) 133–148.
[38] F. Nogueira, Bayesian optimization: open source constrained global optimization tool for Python, 2014, URL: https://github.com/fmfn/BayesianOptimization.
[39] S. Kadioglu, Y. Malitsky, M. Sellmann, K. Tierney, ISAC – instance-specific algorithm configuration, in: H. Coelho, R. Studer, M. Wooldridge (Eds.), Proceedings of the 19th European Conference on Artificial Intelligence (ECAI-10), Frontiers in Intelligence and Applications, vol. 215, 2010, pp. 751–756.
[40] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann, Algorithm portfolios based on cost-sensitive hierarchical clustering, in: Twenty-Third International Joint Conference on Artificial Intelligence, 2013.
[41] C. Ansótegui, J. Pon, M. Sellmann, K. Tierney, Reactive dialectic search portfolios for MaxSAT, in: AAAI, 2017, pp. 765–772.
[42] C. Ansótegui, B. Heymann, J. Pon, M. Sellmann, K. Tierney, Hyper-reactive tabu search for MaxSAT, in: International Conference on Learning and Intelligent Optimization, Springer, 2018, pp. 309–325.
[43] S. Kadioglu, M. Sellmann, Dialectic search, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2009, pp. 486–500.
[44] C. Ansotegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in: I. Gent (Ed.), Principles and Practice of Constraint Programming (CP-09), LNCS, vol. 5732, Springer, 2009, pp. 142–157.
[45] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, K. Tierney, Model-based genetic algorithms for algorithm configuration, in: 24th International Joint Conference on Artificial Intelligence, 2015, pp. 733–739.
[46] C. Ansótegui, J. Pon, M. Sellmann, K. Tierney, PyDGGA: distributed GGA for automatic configuration, in: C.-M. Li, F. Manyà (Eds.), Theory and Applications of Satisfiability Testing – SAT 2021, Springer International Publishing, Cham, 2021, pp. 11–20.
[47] R. Battiti, M. Brunato, A. Mariello, Reactive search optimization: learning while optimizing, in: Handbook of Metaheuristics, Springer, 2019, pp. 479–511.
[48] V. Cicirello, Non-wrapping order crossover: an order preserving crossover operator that respects absolute position 2 (2006) 1125–1132, https://doi.org/10.1145/1143997.1144177.
[49] M. Zaefferer, J. Stork, M. Friese, A. Fischbach, B. Naujoks, T. Bartz-Beielstein, Efficient global optimization for combinatorial problems, in: C. Igel, D.V. Arnold (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2014, ACM Press, New York, NY, 2014, pp. 871–878.
[50] R. Li, M.T. Emmerich, J. Eggermont, T. Bäck, M. Schütz, J. Dijkstra, J. Reiber, Mixed integer evolution strategies for parameter optimization, Evol. Comput. 21 (2013) 29–64.
[51] H. Wang, B. van Stein, M. Emmerich, T. Bäck, Time complexity reduction in efficient global optimization using cluster Kriging, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'17), ACM, Berlin, Germany, 2017, pp. 889–896.
[52] A. Hottung, Y.-D. Kwon, K. Tierney, Efficient active search for combinatorial optimization problems, arXiv preprint, arXiv:2106.05126, 2021.
[53] R.J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Mach. Learn. 8 (1992) 229–256.
[54] S. Ropke, D. Pisinger, An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows, Transp. Sci. 40 (2006) 455–472.
[55] F. Hammami, M. Rekik, L.C. Coelho, A hybrid adaptive large neighborhood search heuristic for the team orienteering problem, Comput. Oper. Res. 123 (2020) 105034.
[56] A.-E. Yahiaoui, A. Moukrim, M. Serairi, The clustered team orienteering problem, Comput. Oper. Res. 111 (2019) 386–399.