A REINFORCEMENT LEARNING APPROACH TO THE ORIENTEERING PROBLEM WITH TIME WINDOWS

A PREPRINT

Ricardo Gama  
Research Centre in Digital Services (CISeD)  
Polytechnic Institute of Viseu, Portugal  
rgama@estgl.ipv.pt

Hugo L. Fernandes  
Rockets of Awesome  
New York City, USA  
hugoguh@gmail.com

July 1, 2021

ABSTRACT

The Orienteering Problem with Time Windows (OPTW) is a combinatorial optimization problem where the goal is to maximize the total score collected from different visited locations. The application of neural network models to combinatorial optimization has recently shown promising results in dealing with similar problems, like the Travelling Salesman Problem. A neural network allows learning solutions using reinforcement learning or supervised learning, depending on the available data. After the learning stage, it can be generalized and quickly fine-tuned to further improve performance and personalization. The advantages are evident since, for real-world applications, solution quality, personalization, and execution times are all important factors that should be taken into account.

This study explores the use of Pointer Network models trained using reinforcement learning to solve the OPTW problem. We propose a modified architecture that leverages Pointer Networks to better address problems related with dynamic time-dependent constraints. Among its various applications, the OPTW can be used to model the Tourist Trip Design Problem (TTDP). We train the Pointer Network with the TTDP problem in mind, by sampling variables that can change across tourists visiting a particular instance-region: starting position, starting time, available time, and the scores given to each point of interest. Once a model-region is trained, it can infer a solution for a particular tourist using beam search. We based the assessment of our approach on several existing benchmark OPTW instances. We show that it generalizes across different tourists that visit each region and that it generally outperforms the most commonly used heuristic, while computing the solution in realistic times.

1 Introduction

The Orienteering Problem and its variants are a family of combinatorial optimization problems with numerous applications, from logistics to telecommunications (Vansteenwegen and Gunawan (2019)). Among them, the Orienteering Problem with Time Windows (OPTW) can be seen as a model for the Tourist Trip Design Problem (TTDP) (Gavalas et al. (2014)). When they plan a trip to a given destination, tourists intend to visit most of their favourite points of interest within a limited time schedule. Usually, the large number of points of interest, located in different places and with different operation hours, makes it arduous and time-consuming to design relevant tourist routes. In the OPTW problem, one has to find the route that maximizes the sum of scores of the visited points of interest considering local and global time constraints.

The OPTW is an NP-hard problem and several heuristics have been proposed to obtain high-quality solutions. The Iterate Local Search (ILS) (Vansteenwegen et al. (2009)) heuristic is one of the most well-known and provides fast and good quality solutions, which allows its incorporation in real-time applications. Gunawan et al. (2015) extended the ILS approach making some point-wise adaptations and including other local search operations, such as swap, 2-opt and replace, which improved the score on standard benchmark instances at the expense of computation time. More recently,
We build a solution in a sequential fashion. For each step of the process, the existing time budget and time windows are allocated to the next visitor to be served. We generate new instances (new instant-tourists) for that instance-region that vary depending on the score provided.

An optimization problem to which the use of these machine learning techniques can be particularly beneficial is the Vehicle Routing Problems (Deudon et al. (2018), Nazari et al. (2018), Kool et al. (2019a), Falkner and Schmidt-Thieme (2020) and Lin et al. (2020)), Knapsack Problem (Gu and Hao (2018)) or the Max-Cut Problem (Gu and Yang (2018)).

To overcome these limitations, Bello et al. (2017) used reinforcement learning to train Pointer Networks for the TSP and the Knapsack Problem. This opens the way to the possibility of tackling a broader set of combinatorial optimization problems. Following that work, other Pointer Network architectural modifications and applications have appeared, e.g., Vehicle Routing Problems (Deudon et al. (2018), Nazari et al. (2018), Kool et al. (2019a), Falkner and Schmidt-Thieme (2020) and Lin et al. (2020)), Knapsack Problem (Gu and Hao (2018)) or the Max-Cut Problem (Gu and Yang (2018)).

An optimization approach to which the use of these machine learning techniques can be particularly beneficial is the TTDP. In a realistic setting, we can expect the variability across instances of the same region to come mostly from the choices made by tourists, and not so much from region specific characteristics like the location of its points of interest or their opening and closing times. This means that, in practice, one needs to repeatedly solve a nearly same instance problem where only a limited set of specific features/parameters may vary, usually in an expected and controlled way. Thus, when access to real data is not possible, the generation of realistic instances for training can be achieved through simulation. This should allow the training with reinforcement learning of a model that can generalize across that variability.

In this paper, we explore the application of Pointer Networks to the OPTW/TTDP problem. We propose a model that takes into account the recurrent nature of the construction process of the sequential solution, using Pointer Networks to better address problems with dynamic time-dependent constraints. This is a recursive attentive model that relies on a Transformer block (Vaswani et al. (2017), Deudon et al. (2018), Kool et al. (2019a)) with dynamical graph self-attention (Veličković et al. (2018)).

2 **Orienteering Problem with Time Windows**

In an OPTW instance, \( \phi \), a set of \( n \) nodes, \( \{v_i\}_{i=1}^n \) with their corresponding coordinates \( x_i \in \mathbb{R}^2 \) are given. Every node \( v_i \) has a positive score or reward, \( r_i \), a visiting time window \( [o_i, c_i] \) with opening time (the earliest a visit can start) and closing time (the time at which a visit has to stop), and duration of visit \( d_i \). Without loss of generality, we can assume that the starting location is \( v_1 \), and \( v_n \) is the ending location for every solution path (also called route). The objective is to find a solution route \( S = (\pi_1, \ldots, \pi_m) \) with the maximum possible sum of scores, without repeating visits, starting the route at or after a given time \( T_{\text{start}} \) and ending it before time \( T_{\text{end}} \) (see Vansteenwegen and Gunawan (2019)).

We build a solution in a sequential fashion. For each step of the process, the existing time budget and time windows constraints have to be taken into account. We start the route in \( v_1 \), initializing the current time \( t^0 \) with the particular instance starting time \( T_{\text{start}} \). After the first initialization step, the path construction process follows iteratively, setting which node, \( v_j \), will be visited next assuming that the following constraints are satisfied:

\[
a_j + \text{wait}_j \leq c_j; \tag{2.1}
\]
where \( a_j \) is the time of arrival at node \( v_j \), \( \text{wait}_j = \max(0, a_j - a_j) \) is the time one has to wait before the visit can start, and \( \Delta_{jn} \) the time it takes to go from node \( v_j \) to node \( v_n \). That is, one must arrive at \( v_j \) before closing time and, at arrival, there needs to be enough time left to visit the \( v_j \) and travel to the last location \( v_n \). After the visit to node \( v_j \), the current time is updated \( t^l = a_j + \text{wait}_j + d_j \) and the construction loop proceeds until \( v_n \) is reached.

At each iteration step \( l \), we define \( \mathcal{A}(v^l, t^l) \) as the set of nodes that are admissible to be visited next. This set is a function of the sub-sequence built until step \( l \), and of the current node \( v^l \) and current time \( t^l \).

The Tourist Trip Design Problem Perspective

An OPTW instance can be interpreted as a TTDP instance, i.e. a particular tourist visiting a particular region. From this perspective, a route can be considered a tourist tour, and the nodes will be points of interest that the tourist might want to visit. Some of the parameters are instance-region specific and tourist invariant, while others are not. Here, we assume that a route’s starting location \( v_1 \), its starting time \( T_{\text{start}} \) and ending time \( T_{\text{end}} \), as well as the scores obtained by the different points of interest are tourist dependent, while the coordinates of the points of interest, their opening and closing times, and the duration of the visit are tourist invariant. We use 3 different sets of benchmark instances (Vansteenwegen et al. (2011), Gavalas et al. (2019), see Section 4.1). Each benchmark instance can be interpreted as a particular tourist visiting a particular instance-region. For each of these instances we can generate new tourists with some variability across the tourist-dependent parameters (see Section 4.2 in Methods). We use these generated tourists or tourist-instance-regions to train, monitor, evaluate and validate our model.

3 Pointer Network Model

Our model is a graph recursive attentive model based on the Pointer Network model architecture (Vinyals et al. (2015)). We conducted a detailed description of the model and of the Pointer Network architecture to explain the particularities of the model, the differences in relation to the other approaches used to solve combinatorial optimization problems using Pointer Network models, and the motivation behind those choices.

For a large set of combinatorial optimization problems, e.g. the TSP and the OPTW, a solution can be constructed sequentially, with elements chosen iteratively one by one. The Pointer Network model architecture (Vinyals et al. (2015)) was designed to address this kind of problem. Its architecture consists of three main blocks (see Figure 3.1): the set encoder block that processes the input set; the sequence encoder block that handles the sequence as it is built; and a third block that consists of a pointing mechanism. Typically, this sequential process is carried out in two distinct phases: an encoding phase and a pointing phase. During the encoding phase, the set encoder block creates a higher-dimensional feature representation of the given input set. Afterwards, the sequence encoding block encodes the solution/sub-sequence constructed so far. In the second phase, both representations are fed into the pointing mechanism to generate a probability distribution over the input set. This probability distribution “points” to which element is more likely to be the best choice as the next sequence element.

![Figure 3.1: The proposed Pointer Network architecture and information flow during an iteration step. (I) - Set encoder block; (II) - Sequence encoder block; (III) - Pointing mechanism; (IV) - Representation vectors computed in the previous iteration step are used recursively in their corresponding functional block.](image-url)
Both the encoding (set and sequence) and the pointing mechanisms can be design in different ways (e.g. Vinyals et al. (2015); Bello et al. (2017); Nazari et al. (2018); Kool et al. (2019a)). Here we use the same sequence encoding and pointing mechanism architecture as Vinyals et al. (2015). The main and substantial differences regarding this and other existing Pointer Network variations lie in the set encoding block. There are three main aspects to highlight when comparing previous Pointer Network studies. Before going into more detail, we will briefly outline them.

New set representation at every iteration.

The first essential aspect is that the set encoder runs in parallel with the sequence encoder and provides an updated vector representation of each node in every iteration step. When building a solution, at each iteration, we recompute the representation of each node (the output of the set encoding block), feeding both dynamic and static features to the set encoding block (Dai et al. (2017)).

It is important to note that this is different from previous proposed Pointer Network models, including from those that also use both static and dynamic features (Nazari et al. (2018); Kool et al. (2019a)). In those works, the static features are only encoded once: in the first step of the solution building process. The dynamic features are used by the sequence-encoder block and pointing mechanism which rely on context-based attention.

In fact, the inclusion of this adjustment in the model architecture allows us to: use recursion in the set encoding block (Transformer with recursion, see below); mask the set of nodes and lookahead connections that are not admissible (Masked self-attention using induced graph structure, see below).

Transformer with recursion. A second aspect is that we use self-attention with recursion (see Section 3.1.2 for details, and “no recursion” in Supplementary materials [A] for model comparison) to try to make the best of the recurrent nature of the solution building process. Intuitively, this change somewhat brings part of the sequence encoding effort to the set encoding block.

Masked self-attention using induced graph structure. A third important aspect is that at each iteration step, and in order to consider the time constraints of the problem, we dynamically compute the set of admissible nodes and a graph structure that can be induced over it (see Section 3.1.3 for detains, and “complete graph” in Supplementary materials [A] for model comparison). We use this graph-structure to apply masked self-attention during set encoding.

3.1 Set Encoding

Formally, given an input set of nodes \( V = \{v_1, v_2, \ldots, v_n\} \), and their corresponding feature vectors \( f = \{f_1, f_2, \ldots, f_n\} \), where \( f_i \in \mathbb{R}^{d_f} \), the first step is to project these vectors into a higher-dimension space through a learnable transformation.

We separate \( f_i \) into static \( f_i^{st} \) and dynamic \( f_i^{dy} \) groups (Nazari et al. (2018)). As the model is building a particular solution, the dynamic features (e.g. time until \( v_i \) opens) may change at each iteration, while the static remain the same throughout that process (e.g. Euclidean coordinates of \( v_i \)). We map each \( f_i \) into \( e_i \in \mathbb{R}^{d_e} \) where \( e_i = [e_i^{st}, e_i^{dy}] \), \( e_i^{st} = \tanh(W^{st}f_i^{st} + b^{st}) \) and \( e_i^{dy} = \tanh(W^{dy}f_i^{dy} + b^{dy}) \), i.e. in a first layer we process the static and dynamic features separately before concatenating them.

These embedded node vectors \( e = \{e_1, e_2, \ldots, e_n\} \) are fed into a Transformer (Vaswani et al. (2017)) with a tweak. This tweak is the introduction of recursion in computing the key of the self-attention sub-block (see details below). The final output of the set encoder and of the transformer is a set \( \{h_i^1, h_i^2, \ldots, h_i^n\} \) representation of \( V \), with \( h_i^k \in \mathbb{R}^{d_e} \). We chose the Transformer architecture both because it is suitable for permutation invariant input sets (the set of points of interest) and because of the good results reported in similar combinatorial optimization problems (Deudon et al. (2018); Kool et al. (2019a)).

3.1.1 Vanilla Transformer

In this section we describe the original transformer architecture before explaining how we introduce recursion. The Transformer architecture consists of a set of stacked layers (here we use 2 layers). Each layer is composed of 2 sublayers or functional components: a multi-head self-attention sublayer followed by a feedforward sublayer.

Given a list of input vectors \( \{e_1, e_2, \ldots, e_n\} \), the self-attention vectors are determined (note that this is the description of the vanilla Transformer, in our model the keys \( k_i \) are computed recursively, see Section 3.1.2) by first computing two linear transformations for each \( e_i \):

\[
k_i = W_k e_i \tag{3.1}
\]

\[
v_i = W_v e_i \tag{3.2}
\]
followed by the determination of the similarity score between each \( e_i \) and each \( k_j \), using scaled dot-product attention:

\[
    u_{ij} = \frac{e_i^T W^k_j}{\sqrt{d_k}}, \quad \forall i, j \in \{1, \ldots, n\},
\]

where \( d_k \) is a scaling constant, \( W_k \), \( W_v \), and \( W_q \) \( \in \mathbb{R}^{d_k \times d_v} \). The variables \( k_i \) and \( e_i \) in Equations 3.1 and 3.2 are usually called key and value, respectively. The output vector \( h^a_i \) is then computed by taking the average of the vector of values \( v \), using a vector \( a_i \), the softmax normalization of the similarity scores vector \( u_i \):

\[
    h^a_i = \sum_j a_{ij} v_j, \quad \text{with} \quad a_{ij} = \frac{\exp(u_{ij})}{\sum_j \exp(u_{ij})}.  
\]

For multi-head attention, the attention mechanics is independently applied, i.e. without weight sharing, multiple times in parallel to the same input. Here we used 8 heads. The vectors resulting from each attention head are concatenated and once again projected into \( \mathbb{R}^{d_e} \) through a linear transformation. This results in the attention representation vectors \( \{h^a_1, h^a_2, \ldots, h^a_n\} \).

The second component of the Transformer layer is then a feedforward fully connected layer:

\[
    h_i = W_2^{ff} \text{Relu}(W_1^{ff} h_i^a + b_1^{ff}) + b_2^{ff}
\]

where \( \text{Relu}(x) = \max(0, x) \), \( W_1^{ff} \in \mathbb{R}^{d_f \times d_e}, W_2^{ff} \in \mathbb{R}^{d_e \times d_f}, b_1^{ff} \in \mathbb{R}^{d_f} \) and \( b_2^{ff} \in \mathbb{R}^{d_e} \).

Both the attention and the feedforward components are followed by a skip connection and layer normalization (Vaswani et al. (2017)). The final output is then the output of the set encoding block, i.e., a representation vector \( h^e_i \in \mathbb{R}^{d_e} \) for each node/point of interest.

### 3.1.2 Transformer with Recursion

We changed the self-attention of the set encoder of previous work (Deudon et al. 2018; Kool et al. 2019a) by making it recursive. In order to make the set encoder block recursive, we changed the self-attention inputs by changing Equation 3.3 to compute each key \( k_i \) at iteration \( l \) using the representation \( h^e_i \) from the previous iteration:

\[
    k_i = W_k h^e_i^{l-1}.
\]

We used \( h^e_i \) for every self-attention layer of the set encoder block. This change forces the self-attention mechanism to be a function of the solution construction state at each iteration step.

### 3.1.3 Graph Masked Self-Attention

In the OPTW, as in many other routing problems, at each iteration step \( l \) of the iterative process of building a solution, the set of admissible locations, \( \mathcal{A}(v^l_i, t^l_i) \), may change significantly. At each iteration step, a different graph structure can be induced over the various elements of the admissible set of locations, \( \mathcal{A} \), leveraging the computation of self-attention during the set encoding (Veličković et al. 2018). One straightforward approach to building such graph is to assume the set of admissible locations as a complete graph and use this graph to perform masked self-attention (see "complete graph" in Supplementary materials [A]). Previous studies (Kool et al. 2019a) have referenced graph attention, however they typically use complete graph representations over the set of admissible nodes and do not take effective advantage of the dynamic graph structure that can be induced. In the next paragraph we explain how we refine that graph and clarify what we mean by masked self-attention.

We define a new graph by considering that the current node or point of interest \( v^l_i \) is connected to node \( v_j \) if \( [v^l_i, v_i, v_j, v_n] \) is a feasible sequence, where \( v^l_i \) is the current node and \( v_n \) is the ending location. This is essentially a one-step lookahead search. At every iteration step \( l \), we determine the adjacency matrix \( Ad^l \) of the graph. When performing self-attention, the scores \( u_{ij} \) are computed using Equation 3.3 only if \( Ad^l_{ij} = 1 \), and are set to zero otherwise (i.e. are masked). By masking not only the immediate non-admissible nodes, but also one-step lookahead paths, it is reasonable to expect better representation vectors for each node or point of interest.

Note that using graph masked self-attention and dynamic features requires updating the set representation at each iteration step. Performing set encoding only at the first step/iteration would make the model ignore these relevant pieces of information.
3.2 Sequence Encoding

The main objective of this functional block is to compute a vector representation of the subsequence that is being built iteratively. We denote this vector by $h_{d,l}$, with $h_{d,l} \in \mathbb{R}^d$.

We call this block the "sequence encoding" block. The name might generate some confusion since, in NLP-translation/seq-2-seq applications where the Pointer Network architecture originated, this block is typically named the "decoding" block as it is translating/decoding into the target language. For the OPTW, and other combinatorial optimization problems, "sequence encoding" seems a more suitable name considering its role in the overall architecture.

As in Vinyals et al. (2015) we chose a Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber (1997)) for sequence encoding. Since we compute a new set encoding at each iteration $l$, together with the hidden state vector of the previous step, $h_{d,l-1}$ we can give as input to the LSTM the vector representation of the current point of interest computed by the set encoder in the current iteration step, $h_{e,l}$. By relying on an up-to-date iterative encoding of each point of interest, we expect to obtain a better encoding of the sequence built up to step $l$.

3.3 Pointing Mechanism

On top of the set and sequence encoder blocks, lies the pointing mechanism (see Figure 3.1). At each iteration step $l$, it takes as input both the set encoding of each node $\{h_{e,l}^1, h_{e,l}^2, \cdots, h_{e,l}^n\}$ and the latest hidden state $h_{d,l}$ of the sequence encoder’s LSTM, and computes a probability distribution over the set of points of interest. This probability vector can then be used for selecting the point of interest $v^l$.

To get to this probability vector we start by computing an importance score for each element of the input set using additive attention (Bahdanau et al. (2015))

$$
    u^l_j = \begin{cases} 
    w^T \tanh(W_1 h_{e,l}^j + W_2 h_{d,l}), & \forall j : v_j \in A^{l-1} \\
    -\infty, & \text{otherwise}
    \end{cases}
$$

(3.7)

where $w \in \mathbb{R}^d$ and the matrices $W_1 \in \mathbb{R}^d \times d_e$ and $W_2 \in \mathbb{R}^d \times d_d$ are learnable parameters. Points of interest that cannot be visited, i.e. $v_j \notin A^{l-1}$, are masked (Bello et al. (2017); Deudon et al. (2018); Kool et al. (2019a)). In order to control the range of the logits, $u^l_j$ is subsequently transformed by

$$
    \tilde{u}^l_j = \begin{cases} 
    C \tanh(u^l_j), & \forall j : v_j \in A^{l-1} \\
    -\infty, & \text{otherwise}
    \end{cases}
$$

(3.8)

where $C$ a hyper-parameter (Bello et al. (2017)). Finally we obtain the probability distribution over the points of interests after a softmax normalization of $\tilde{u}^l_j$

$$
    p_l(v^l_j | A^{l-1}, S^{l-1}) = \frac{\exp(\tilde{u}^l_j)}{\sum_{j'} \exp(\tilde{u}^l_{j'})}.
$$

(3.9)

Using this probability distribution, we can choose the next point of interest to be visited and after its selection, the iterative loop proceeds until a stopping condition (i.e. reaching $v^n$) is met.

4 Methods

4.1 Benchmark Instances

We use three groups of benchmark instances: Solomon, Cordeau and Gavalas. All these instances have best-known results documented in recent publications (Karabulut and Tasgetiren (2020); Gavalas et al. (2019)) and represent a diverse set, with different distributions of points of interest (Figs. C.1, C.2 and C.3), scores and duration of visit (Figs. C.4, C.5 and C.6) and schedules (Figs. C.7, C.8 and C.9).

The Solomon and Cordeau instances are groups of established benchmarks in the OPTW literature (Vansteenwegen et al. (2011)). The Solomon group was originally adapted from a set of vehicle routing problems with time windows and is composed of 56 instances, all with 100 points of interest. The Cordeau group, includes 20 instances, ranging from 48 to 288 points of interest.
The Gavalas instances (Gavalas et al. 2019) were designed to be more representative of real TTDP problems. In particular, their times lay within a 24-hour (1440 minutes) time range and they were generated with the assumption that a tourist will attach higher value, on average, to a point of interest that requires more time to visit, i.e., they all have a correlation between the scores and the duration of the visit of each point of interest (see Figure C.6). The Gavalas group includes instances for the more general problem of Team OPTW, which can be framed as a tourist problem if we allow the tourist to use more than one day to visit the instance-region. Here, since we are only addressing the OPTW (and not the TOPTW), we consider only the 33 Gavalas instances for which the tourist has a single day available. These instances range from 100 to 200 points of interest (see Section C in Supplementary materials and Vansteenwegen and Gunawan (2019) for further characterization of all benchmark instances).

The distances between locations are rounded to the first decimal place for the Solomon instances and to the second decimal place for the Cordeau and Gavalas instances. For most of our analysis, we focus on a subset of each of the groups, which we dubbed sub-Cordeau (8 instances), sub-Solomon (12 instances) and sub-Gavalas (8 instances), making it a total of 28 instances (see Section 4.6).

4.2 Generated Instances

Some of the parameters of an instance are tourist dependent and some others are invariant across tourists. We can look at each instance as the representation of a given tourist visiting a given region. Each tourist has a particular taste (scores), and preferences regarding starting locations and starting and ending times. Here we assume that the coordinates of the points of interest, the duration of the visit and the opening and closing times are instance-region parameters and remain unchanged regardless of the tourists.

Having a model for a particular instance-region means having a model that works for all tourists that might visit that region. For that reason, for each benchmark instance we generate new tourists, i.e., new instances with the same instance-region parameters but different tourist parameters. We use these generated instances for training, monitoring and validation (see Sections 4.4 and 4.5). The choice of hyper-parameters for sampling was not done to specifically optimize generalization for the benchmark tourist-instance, but instead to generalize a reasonable range of tourists’ parameters while maintaining the same hyper-parameters across groups of instances.

Generate Starting (and Ending) Location

We sample uniformly a new starting location from the square for Cordeau and Gavalas instances. In all the benchmark instances, the tour ends in the same position as it starts, and we assume the same for the generated instances.

Generate Starting and Ending Times

Let us name \( T_{\text{start}}^b \) and \( T_{\text{end}}^b \) the starting time of the tour and the upper-bound on the tour’s end time, respectively, for the tourist in the benchmark instance. We want to sample new times \( T_{\text{start}}^g \) and \( T_{\text{end}}^g \) for a new generated tourist. We chose this sampling empirically because it seems realistic enough and works for the 3 groups of benchmark instances. Most importantly we wanted it to span a range that is diverse enough and that would include the \( T_{\text{start}}^b \) and \( T_{\text{end}}^b \).

We start by defining the duration of a day for a benchmark-instance as the maximum time stamp of that instance: 
\[
T_{\text{day}}^b := \max \{ T_{\text{end}}^b, \max_{i=1..n} \{ c_i \} \}.
\]

Then we normalize all times so that the duration of a day is 24 (as in 24 hours). Finally we sample \( T_{\text{start}}^g \) as
\[
T_{\text{start}}^g \sim U(T_{\text{start}}^b - 4, ub)
\]
where
\[
ub = \min \{ 15, T_{\text{end}}^b + 4 \},
\]
i.e. the tour starts 4 "hours" at most before the benchmark-tourist, and before time 15 (3pm) unless the tourist in the benchmark-instance ends tour 4 "hours" before 3pm, in which case \( T_{\text{end}}^b + 4 \) is the upper bound \( ub \). We sample \( T_{\text{end}}^g \) uniformly:
\[
T_{\text{end}}^g \sim U(lb, T_{\text{end}}^b + 4)
\]
where
\[
lb = \max \{ 12, T_{\text{start}}^g + 4 \},
\]
i.e. \( T_{\text{end}}^g \) is at most 4 "hours" after the benchmark-tourist, not before "noon" and there also must be at least 4 hours left for the tour. After the sampling, we divide by 24 and multiply back by the duration of the day for the benchmark-instance and round it to the closest integer (see Figure 4.1).
The Gavalas instances were created to reflect essential characteristics of a realistic TTDP. In particular, the authors assumed that the scores that a tourist gives to each point of interest is correlated with the time it takes to visit that point of interest. The authors do not specify how this correlation was generated (Gavalas et al. (2019)) but, in an attempt to improve learning, our aim was to generate instances with the same property. For that reason, when sampling scores we tried two different sampling schemes for the Gavalas instances: a uniform sampling scheme and a non-uniform or correlated one.

In the uniform scheme the scores are uniformly sampled from $[1, 1.1 \times S_{max}^b]$ where $S_{max}^b$ is the maximum score for the benchmark instance. This is the scheme used for the Solomon and Cordeau generated instances. In the correlated sampling scheme, the scores are generated according to:

$$s_i \sim \mathcal{N}(S_{max}^b \frac{d_i}{d_{max}}, 100)$$

and clipped to be in the $[1, 1.1 \times S_{max}^b]$ interval, where $d_{max} = \max_{j=1,\ldots,n} \{d_j\}$ is the maximum duration of visit across all points of interest of that instance. Note that $1.1 \times S_{max}^b$ is an upper-bound on the scores of the generated instances, which we will use as a normalization factor for score-related features. For visualizations of the correlation between score and duration of visit for every benchmark instance see Figures C.3, C.5 and C.6.

These generated instance-tourists are used to train the model: during training they are generated on the fly for the given instance-region the model is being trained for (see Section 4.4). For each instance of the three benchmark groups, we have also created a fixed validation set consisting of 64 generated instances. These fixed validation sets are used to monitor each model’s training and evaluate its performance.

### 4.3 Input Features

The first encoding block of the neural network has a set of feature vectors as input: one feature vector for each point of interest (see Section 3.1). These features can be divided into 2 groups: the static features and the dynamic features.

The static features remain constant during the iterative process of building a solution. Here, we use 7 straightforward features retrieved directly from the instance data, namely the point of interest’s Euclidean coordinates, visiting duration, opening and closing time and score, and also the upper bound on the finishing time of the trip, $T_{end}$. The Euclidean
coordinates are mapped using min-max scaling into unit square $[-1, 1] \times [-1, 1]$. The scores are normalized by $1.1 \times S_{\text{max}}^b$ and all the time features are normalized by $T_{\text{max}}$. Note that these normalization constants are the same across tourists for a same instance-region.

The dynamic features can change at every iteration step. We designed 8 features that are functions of the current time, $t^l$, and current point of interest, $v^l$: for each point of interest, the time left until the opening time and time left until closing time; the fraction of time elapsed since the $T_{\text{start}}$; the fraction of the time left to $T_{\text{end}}$. To these features we add the same set of 4 features but assuming we have traveled from the current position to the point of interest that the feature corresponds to (remember that there is one feature vector for each point of interest), i.e., adding to $t^l$, the travel time from $v^l$ to $v_i$. All the dynamic time features are normalized by the maximum time available for the tourist to perform the tour, i.e. $T_{\text{end}} - T_{\text{start}}$. Thus, contrary to the static time features, this normalization constant is tourist dependent.

4.4 Training

We train the neural network model using reinforcement learning.

Objective Function

For a given instance $\phi$, the Pointer Network model, with parameterization $\theta$ defines, at each iteration step, a stochastic policy from which the next visiting location can be sampled. We can use this policy iteratively until the ending location is reached and obtain a solution/route $S = (\pi_1, \ldots, \pi_m)$. For simplicity we can represent this by $S \sim p_\theta(\cdot|\phi)$.

The total route probability can be determined using the chain rule of probability:

$$p_\theta(S|\phi) = \prod_{l=1}^m p_\theta(\pi_l|\phi, A^{l-1}, S^{l-1}).$$

We want to determine $\theta$ such that we can sample high total score solutions from its policy with high probability.

For a sample solution $S$, the total score is given by $R(S) = \sum_{l=1}^m r(\pi_l)$ and our objective function is defined as the expected total score, that for instance $\phi$ is given by:

$$J(\theta|\phi) = \mathbb{E}_{S \sim p_\theta(\cdot|\phi)}[R(S)]$$

REINFORCE Algorithm

We maximize $J(\theta)$ using gradient ascent, and resort to REINFORCE algorithm (Williams (1992)) to estimate the gradients (see Algorithm 1).

**Algorithm 1: REINFORCE algorithm**

```
Input: training set $\Phi$, batch size $B$
1 Initialize network parameters $\theta$;
2 while training not finished do
3    sample instance $\phi$ from $\Phi$;
4    for every $b \in \{1, \ldots, B\}$ do
5        $l \leftarrow 1$
6        while terminal node not reached do
7            sample $\pi_l$ from $p_\theta(\cdot|\phi, A^{l-1}, S^{l-1})$;
8            $l \leftarrow l + 1$
9        end
10        $R \leftarrow \frac{1}{B} \sum_{b=1}^B R(S_b)$;
11        $g_\theta \leftarrow -\frac{1}{B} \sum_{b=1}^B (R(S_b) - R) \nabla_\theta \log p_\theta(S_b|\phi)$;
12        update $\theta$ using $g_\theta$;
13 end
```

Typically, when using the REINFORCE algorithm, each batch is composed by $B$ different (generated) instances. In our approach, however, each batch comprises a single generated instance and thus $B$ independent solutions are sampled for that same single tourist-instance. This allows a straightforward baseline estimation, thus avoiding the need to maintain
a moving average estimate of the baseline, obtain an estimate of it through a greedy roll-out computation, (Kool et al. (2019a)), or having to rely on a critic neural network, (Bello et al. (2017); Vaswani et al. (2017)). In concrete terms, we determine the batch average total score, \( R = \frac{1}{B} \sum_{k=1}^{B} R(S_k) \), and use it as our baseline to reduce gradient estimation variance. This approach is a particular case of the multiple sampling with replacement method (Kool et al. (2019b) with \( k = 1 \)) where the number of solution samples equals to the batch size.

Other Training Details

We train all models with the Adam optimizer (Kingma and Ba (2015)) and use a batch size of \( B = 32 \). We use 500,000 epochs for models trained from scratch and 50,000 epochs for the models that are fine-tuned (see 4.6). For models trained from scratch all parameters are initialized with the Xavier uniform initialization (Glorot and Bengio (2010)), and an initial learning rate of \( 10^{-4} \) decaying every 5000 steps by a factor of 0.96, until a minimum of \( 10^{-5} \). For fine-tuning we use a fixed learning rate of \( 10^{-5} \). Both the first hidden and the first cell state vectors of the sequence encoder’s LSTM are treated as learnable parameters and initialized from \( \mathcal{U}(-\frac{1}{\sqrt{d_h}}, \frac{1}{\sqrt{d_h}}) \).

We set the hyper-parameter \( C = 10 \), in Equation 3.3 as suggested in previous literature (Bello et al. (2017)). We tried to set it to be a learnable parameter without much success in the few attempts and confirmed empirically the quality of the suggested value.

We use 2 transformer blocks with 8 heads in the multi-head self-attention. We use \( d_h = d_e = d_d = 128 \) and \( d_{ff} = 256 \).

4.5 Inference

Inference is the process of building a solution for a particular instance-tourist from the model (or network policy) trained in that instance-region. We use four different solution construction strategies, namely: sampling, greedy, beam search, and active search. We use sampling during training in the REINFORCE algorithm and greedy search to monitor the evolution of learning. In a sampling strategy, for each step, we sample from the probability vector that the model outputs. In a greedy strategy, for each time step, we select the node/point of interest for which the model gives the highest probability, that is \( \pi_t = \arg \max_{v_i \in \mathcal{A}^{t-1}} p_{\theta}(\cdot | \phi, \mathcal{A}^{t-1}, \mathcal{S}^{t-1}) \).

Beam Search

Beam search is a search heuristic that approximately maximizes the total route probability under the network policy \( p_{\theta}(\cdot | \phi) \). It keeps a record, at each time step, a list of the \( n_b \) most promising partial solutions sequences. From these candidates, beam search considers all admissible next visiting locations, and selects the best top-\( n_b \) of them. When every beam reaches the final location, the solution with higher score is selected. When \( n_b = 1 \), beam-search heuristic reduces to a simple greedy search. For beam search inference we show performance for \( n_b \) ranging from 1 up to a maximum of 128 and finally consider performance and processing times for a maximum of 128 beams. Note that the number of beams cannot be higher than the number of nodes in the instance.

Active Search (with Beam Search)

Active Search (Bello et al. (2017)) optimizes the network policy for the given tourist-region instance, i.e., given an instance \( \phi \) we can retrain the model’s weights directly on that instance. The vanilla version uses greedy inference, here we use beam search (see Algorithm 2).

Even though it is considered an inference method, active search can be seen as a way to achieve training/fine-tuning of the region-model for a specific tourist. For active search, we use 128 epochs and a maximum of 128 beams for the beam search that follows active search.

4.6 Experiments

In addition to exploring different kinds of inference, we also explore different training schemes. All the experiments conducted allowed us to obtain results for at least a subset of 28 benchmark instances (or 3 subsets, 8 of Cordeau, 8 of Gavalas and 12 of Solomon). The instances of these subsets are: Solomon: c101, c102, r101, r102, rc101, rc102, c201, c202, r201, r202, rc201, rc202; Cordeau: pr01, pr02, pr03, pr04, pr11, pr12, pr13, pr14; Gavalas: t101, t105, t114, t117, t201, t202, t203, t204. We chose these subsets because we wanted to have results for at least 8 instances of each group and at the same time have a balanced representation of the diversity within each group. In all models/experiments we use beam search for inference with up to a maximum of 128 beams.
Algorithm 2: Active Search algorithm

Input: instance $\Phi$, network parameters $\Theta$, batch size $B$

1 while $epoch \leq epoch_{max}$ do
2     for all $b \in \{1, \ldots, B\}$ do
3         $l \leftarrow 1$
4         while terminal node not reached do
5             sample $\pi_l$ from $p_\theta(\cdot|\Phi, A_{l-1}^b, S_{l-1}^b)$;
6             $l \leftarrow l + 1$;
7         end
8     end
9     $R \leftarrow \frac{1}{B} \sum_{b=1}^{B} R(S_b)$;
10    $g_\theta \leftarrow -\frac{1}{B} \sum_{b=1}^{B} \left( R(S_b) - \overline{R} \right) \nabla_\theta \log p_\theta(S_b | \Phi)$;
11    update $\Theta$ using $g_\theta$;
12 end
13 $S \leftarrow$ Beam Search($\Phi, \Theta$);

Models Trained from Scratch

For the main result, we train models from scratch for 500,000 epochs for each of the subset of 28 instances. Then, we look at performance with "$\text{model}+\text{as}$" and without "$\text{model}$" 128 epochs of active search. We also look at the performance of "$\text{model}$" at 50,000 epochs of training: we denote this model by "$\text{st}$".

Transfer Learning

For each instance group, we also use the instances in the subset of 28 instances as leave-out sets for evaluating the feasibility of using transfer learning: we train a global model on all instances except for the instances in the leave-out set and then evaluate its performance regarding those left-out instances. We called this model "$\text{tl}$". Furthermore, we evaluate the performance of fine-tuning that global model on each of those left-out instances of the subset. Fine-tuning happens for 10% of regular training, i.e. 50,000 epochs. We called these fine-tuned models "$\text{ft}+\text{tl}$". While training a global model, i.e., a model in several instances at the same time, the instance chosen for each batch in the REINFORCE algorithm is generated from one randomly chosen benchmark instance of the training set of instances.

General global Model

Finally, we also train a global model on all instances simultaneously and then fine-tune it for each of the instances. This was done to investigate the feasibility of warming up a general model when we have a lot of different instance-regions to train.

All experiments were performed on a 12 core CPU at 3.5GHz (AMD Threadripper 1920X) with 64 GB of RAM and an Nvidia GeForce GTX 1080 Ti GPU.

4.7 Metrics and Statistical analysis

To quantify performance, we use total score of the solution, i.e., the sum of the scores/rewards. For each generated instance group, the reported score is the mean value of the average score across the 64 instances of the validation sets within that group.

We compare our model’s performance to the Iterate Local Search (ILS) heuristic (Vansteenwegen et al. (2009)), as it presents one of the best trade-offs between solution quality and execution time. We also report the score gap to the best-known solution in the benchmark instances. We define score gap to a baseline model as

$$\frac{\text{score}_{\text{baseline}} - \text{score}_{\text{model}}}{\text{score}_{\text{baseline}}} \times 100$$

We use bootstrapping to obtain a 95% confidence intervals for gap to ILS and gap to best known. We sample with replacement the benchmark instances or instant-regions 10000 times. For determining the p-values in pairwise comparisons of scores we use the non-parametric one-sided Wilcoxon signed-rank test.
5 Results

We investigate the performance of an attentive deep reinforcement learning approach to solving the Orienteering Problem with Time Windows (OPTW). We use a Pointer Network architecture and the REINFORCE algorithm to estimate gradients during training. We evaluate the performance of greedy inference as well as that of beam search and active search. We address the problem with the Tourist Trip Design Problem (TTDP) application in mind, but our model’s performance and inference times do not limit it to that application. We consider 3 groups of benchmark tourist-instance-regions (the “benchmark” instances) and generate new tourists for each of the instance-regions (the “generated” instances) in a way that mimics variations expected in the TTDP applications (i.e., the variability across tourists). In concrete terms, from each benchmark instance we sample new instances (new tourists for that instance’s region) with different route starting positions, different route starting and ending times, and different tourist-specific scores/preferences for each point of interest. Then, we compare performance with a well-established heuristic, the Iterated Local Search (ILS) algorithm, on both the benchmark and generated instances, and quantify inference speeds. Finally, we explore the practicality and performance of transfer learning and fine-tuning training schemes.

The model is able to learn and achieve production-level performance

First we wanted to know if the model can learn a particular instance-region when trained from scratch. We evaluate training on a subset 28 instances (n=12 for Solomon, n=8 for Cordeau and n=8 for Gavalas, see Section 4. Methods for details). We find that the model is able to learn for each set of instances on both the generated instances (Fig. 5.1A blue) and also that it generalizes to the benchmark instances (Fig. 5.1B blue) during training. At the end of training the performance (using greedy inference) is already in line with ILS. We find that beam search significantly improves performance (Fig. 5.1A and B red) and that it is generally able to significantly outperform ILS (Table 5.1) with inference times below half a second. In fact, we find that already with only 10% of training (50,000 epochs) the model already has higher scores than ILS in the generated instances ($p = 2.6E−06$, one-sided Wilcoxon signed-rank test, see “ILS→st” in Figure 5.4). Our model is able to learn and achieve competitive production level performance in both scores and execution times.

![Figure 5.1](image-url)
A Reinforcement Learning Approach to the Orienteering Problem with Time Windows

A P

Table 5.1: Average performance (average score, gap to ILS and gap to best-known) and inference times of beam search inference with a maximum number of 128 beams for models trained from scratch for 500,000 epochs on each of the instance-regions in the subset of instances (12 Solomon, 8 Cordeau and 8 Gavalas). The generated instances include n=64 generated instances for each of template benchmark instance-regions. The 95% confidence intervals are computed using bootstrap (sampling instance-regions with replacement). For performance on each of the 28 individual benchmark instances and average performance for each individual instant-region see "model" in Tables B.1 and B.2 in Supplementary materials.

Active search inference improves performance at the cost of speed

Beam search inference produces good results in less than half a second. However, if the user is willing to wait a couple of minutes for a better solution, active search (Bello et al. (2017), see Algorithm 2) can be an appropriate inference solution. In active search the model is fine-tuned on for a particular tourist-region instance, improving its policy for solving the problem for that tourist. We apply active search to the models trained from scratch on the same subset of 28 instances, before applying beam search. Aiming for realistic off-line computation times (under 2 minutes), we applied 128 epochs of active search for each benchmark and generated tourist-region instance. We found that indeed while it significantly slower (it is considerably slower than beam search (see Figure 5.2 and Table 5.2) it improves the scores ($p = 1.6E−05$, one-sided Wilcoxon signed-rank test; see "model+as" in Figure 5.4, and Table 5.2). Active search is slower than beam search but can give a justified performance boost for situations where an immediate response is not necessary.

Figure 5.2: Average score in the generated instances as a function of inference time. Results are for 64 generated instances/tourists for each of the 28 benchmark template instance-regions in the subgroup of instances trained individually (12 Solomon, 8 Cordeau and 8 Gavalas). Scores are for inference using beam search changing the maximum number of beams from 1 to 128 (red) and for active search (yellow) with 1 and up to 128 epochs followed by beam search with a maximum of 128 beams. The x-axis is capped at 5 seconds for visualization purposes. See Table 5.2 for inference times with 128 epochs of active search.

Transfer learning and fine-tuning

Next, we investigate the possibility of speeding up training by fine-tuning a model trained on another set of relatively similar instance-regions. This could be relevant in practice, if there is some change to the current instance-region, e.g., opening times changes, new points of interest opened and others closed, etc., and we do not want to train the model from scratch. It could also happen in practice, if one wants to train for a new instance-region while potentially taking advantage of an already warmed-up model. We train on all instances of each of the 3 sets of instances while leaving out
We find that with transfer learning we are able to reach good levels of performance in all subgroups (see Figure 5.3). While each for benchmark instances there is a single tourist per instance. See Figure 5.4 for more granularity and further analysis on the left panel data.

We find that with transfer learning we are able to reach good levels of performance in all subgroups (see Figure 5.3). Even without any fine-tuning, it achieves higher scores than ILS ($p = 2.9E - 06$, one-sided Wilcoxon signed-rank test) and it is better than training on each specific instance from scratch for just 50,000 epochs ($p = 1.4E - 03$, one-sided Wilcoxon signed-rank test, see "st→tl" in Figure 5.4). We find also that fine-tuning for 50,000 epochs further improves the transfer learned model ($p = 1.1E - 05$, one-sided Wilcoxon signed-rank test, see "tl→tl+ft" in Figure 5.4) to performance levels closer to training from scratch even if still lower ($p = 8.6E - 04$, one-sided Wilcoxon signed-rank test, see "tl+ft→model" in Figure 5.4).

### Importance of representative generated instances

In generating new instances, we aimed at mimicking the kind of variability expected across tourists for a particular instance-region. We did this while trying to use the same hyperparameters across groups of instances (see Section

Table 5.2: Average performance of inference using active search with 128 epochs followed by beam search with a maximum number of 128 beams. Results are for the subset of instance-regions trained from scratch (n=12 for Solomon, n=8 Cordeau and n=8 for Gavalas). The generated instances include n=64 generated instances for each of template benchmark instance-regions. The 95% confidence intervals are computed using bootstrap (sampling instance-regions with replacement). For performance on each of the 28 individual benchmark instances and average performance for each individual instant-region see "model+as" in Tables B.1 and B.2 in Supplementary materials.

| Instance Group | Best-Known (bk) | ILS | Model Scores | Gap to ILS | Gap to ILS [95% CI] | Gap to bh | Gap to bh [95% CI] | Time [s] | Time [95% CI] |
|----------------|-----------------|-----|--------------|------------|-------------------|----------|-------------------|----------|---------------|
| Solomon [generated] | - | 429.99 | 439.76 | -3.2% | [-3.9%, -2.56%] | - | - | 27.069 | [19.1, 35.2] |
| Cordeau [generated] | - | 250.60 | 287.32 | 5.14% | [3.03%, 4.20%] | - | - | 39.252 | [23.1, 54.4] |
| Gavalas [generated] | - | 299.99 | 334.50 | -11.73% | [-14.62%, -9.86%] | - | - | 49.853 | [37.5, 61.5] |
| Solomon [benchmark] | 575.58 | 558.83 | 201.00 | -2.16% | [-3.77%, -0.82%] | 0.50% | [0.03%, 1.16%] | 47.682 | [33.4, 62.5] |
| Cordeau [benchmark] | 428.00 | 401.62 | 420.25 | -4.38% | [-7.74%, -1.51%] | 1.64% | [0.65%, 2.67%] | 61.512 | [41.9, 84.1] |
| Gavalas [benchmark] | 101.75 | 307.50 | 316.12 | -2.73% | [-4.94%, -1.21%] | -1.65% | [-2.61%, -0.64%] | 48.144 | [31.6, 65.5] |

Figure 5.3: Performance of the different training/inference strategies. Gap to ILS in generated (left panel) and benchmark (middle panel) instances, and gap to best-known in benchmark instances (right panel), within the subset of 28 instances (12 Solomon, 8 Cordeau and 9 Gavalas), for the different training or inference strategies: training for 50,000 epochs from scratch (st); transfer learning of a model trained simultaneously during 500,000 epochs in all instances within each group except for those in the subset of 28 instances (tl), fine-tuning the transfer learning model for each instance during 50,000 epochs (tl+ft), training from scratch on each individual instance-region for 500,000 epochs (model), "model" with added 128 epochs of active search inference (model+as) before beam search. All models use beam search with a maximum of 128 beams. In the "generated" panel (left) we consider 64 fixed generated tourists for each instance-region, while each for benchmark instances there is a single tourist per instance. See Figure 5.4 for more granularity and further analysis on the left panel data.

Figure 5.4: Performance levels for the different strategies with more granularity and further analysis on the left panel data.
A Reinforcement Learning Approach to the Orienteering Problem with Time Windows

Figure 5.4: Improvement in score in generated instances for different training/inference strategies. Each data point is the average of 64 generated tourists for each instance-region (see Figure 5.3 left panel). Inference in all models uses beam search with up to a maximum of 128 beams. Training for 50,000 epochs from scratch (st); transfer learning of a model trained simultaneously during 500,000 epochs in all instances within each group except for those in the subset of 28 instances (tl), fine-tuning the tl model for each instance during 50,000 epochs (tl+ft), training from scratch on each individual instance-region for 500,000 epochs (model), "model" with added 128 epochs of active search inference (model+as) before beam search. For example, "ILS→st" represents the average score improvement from the ILS algorithm to the model trained for just 50,000 epochs across the 64 fixed generated instances for each instance-region in the subset of the 28 (12 Solomon, 8 Cordeau, 8 Gavalas). The p-values are from one-sided Wilcoxon signed-rank test.

4.2 in Methods). However, in the Gavalas set, we are given some information about the generative process (Gavalas et al., 2019). In concrete terms, we know that the score of each point of interest is correlated with how long it takes to visit that point of interest. We used that information when generating new tourists for each instance-region of the Gavalas set. We found that ignoring that information leads to weak performances on the real instances (see Figure 5.5 B vs Figure 5.1 B 3rd column). It also leads to weaker performances on the generated instances (see Figure 5.5 A vs Figure 5.1 A 3rd column) even though it still manages to outperform the ILS algorithm for these. This probably happens because forcing that correlation reduces the dimensionality of the space of generated instances and therefore the model trains more efficiently. On the other hand, since the distribution of instances used during training is closer to the benchmark instances, the model generalizes better for that instance. Having a better generative process or sampling method improves the performance of the model.

**Fine-tune a global model for every instance**

Finally, we explore another possible practical scenario in which the OPTW has to be solved for a good number of different instance-regions. If any cost, energy, or time restriction presents itself as an impediment, it may be wiser not to train a model from scratch for each instance-region. Instead, we can take advantage of the fact that transfer-learning and fine-tuning produce good results, and in a similar way obtain a model for each instance-region in a reasonable amount of time. With this aim, we start by training one global model for each instance group (without leaving any instances out) and then fine-tune it for 10% (50,000) of the epochs on each individual instance-region. By global we mean that it is simultaneously trained on all the instance-regions of that group. We thus obtain one fine-tuned model for each instance-region with considerable savings in resources. We find that both on the subset of 28 instances (Table 5.3) as well as when looking at all instances (Table 5.4) this model and training scheme is able to outperform ILS.
A Reinforcement Learning Approach to the Orienteering Problem with Time Windows

Figure 5.5: Evolution of model performance during training using greedy inference (blue) and post-training model performance using beam search for inference (red) with up to a maximum number of 128 beams, in both benchmark and generated instances for the three sub-set of n=8 Gavalas instance-regions using uniform sampling of rewards/scores (for Gavalas with correlated sampling see Figure 5.1 right panels). Evolution of the mean validation score during training and model performance using Beam Search for inference. A. Average score gap to ILS on the generated instances (n=64 generated instances per instance-region). B. Average score gap to best-known and ILS-score gap to best-known on the benchmark instances. Inference during training (blue) uses greedy inference. Shaded area represents 90% confidence intervals (bootstrap instance-regions).

Table 5.3: Average performance on the subset of instances (n=12 for Solomon, n=8 Cordeau and n=8 for Gavalas) of a global model trained on all instances and then fine-tuned during 50,000 epochs for each instance-region. Inference using beam search with a maximum number of 128 beams. The generated instances include n=64 generated instances for each of template benchmark instance-regions. The 95% confidence intervals are computed using bootstrap (sampling instance-regions with replacement). For detailed performance on each individual benchmark instance and average performance for each instance-region see Tables B.3, B.4, B.5, B.6, B.7 and B.8 in Supplementary materials.
While we use beam search for inference, we use stochastic sampling as an exploration strategy. We briefly explored the possibility of having access to real data, e.g., tours and tourist feedback, could lead to models that enhance route usability and could better address a tourist’s preferences. The formal optimal solution of the abstract optimization problem may not always be the best practical solution in a real-world setting. The ideal tour may depend on many other features not considered in the abstract model.

To start with, we use blocks of transformers (Deudon et al. 2018) that being permutation invariant, unlike RNNs (Nazari et al. 2018), are well suited for set encoding. Furthermore, we apply set encoding in a dynamic or iterative way. In particular, we use dynamical features that are updated at each iteration and we use one step lookahead to construct a graph representation of the admissible points of interest and use its adjacency matrix for self-attention masking. Importantly, we introduce a change in the transformer architecture: we add recursion by making the attention sub-block dependent on the previous iteration step. While these changes seem to improve the model’s performance in this and other applications of the PNNs.

We have shown that we can get production-level performance and inference times. While ultimately the inference time of the epochs produces higher scores than the ILS algorithm and we have explored several ways of improving training speed and performance. In particular, we show that having a better generative model of tourists or a better sampling strategy improves training, and we also show, using different practical training scenarios, that using warmed-up models, transfer learning and fine-tuning works and speeds-up training.

The first application of PNNs for solving combinatorial optimization problems (Vinyals et al. 2015) used supervised learning on data generated by algorithmic/heuristic approaches instead of reinforcement learning. This limits performance to how good the heuristics are. It would be interesting to see if such strategy could be used to warm-up a model for a reinforcement learning approach.

A hybrid supervised and reinforcement learning approach could also benefit from access to real data. Our work shows that neural networks can be used in real world applications of the TTDP problem. The possibility of having access to real data, e.g., tours and tourist feedback, could lead to models that enhance route usability and could better address a tourist’s preferences. The formal optimal solution of the abstract optimization problem may not always be the best practical solution in a real-world setting. The ideal tour may depend on many other features not considered in the OPTW, which could be implicitly addressed in a machine learning approach that is capable of leveraging real data. The exploration strategy in a reinforcement learning approach can also have a big impact on performance and learning speed. While we use beam search for inference, we use stochastic sampling as an exploration strategy. We briefly explored using stochastic beam search but obtained worse results. It would be interesting to try different exploration strategies, like for instance a Monte Carlo tree search (Browne et al. 2012). Another aspect of our exploration strategy is that

| instance group | best-known | ILS | model score | gap to ILS | gap to ILS [95% CI] | gap to bk | gap to bk [95% CI] |
|---------------|------------|-----|-------------|-----------|----------------------|---------|-------------------|
| Solomon [generated] | - | 480.50 | 487.26 | -2.63% | [-3.23% , -2.06%] | - | - |
| Cordeau [generated] | - | 279.16 | 293.99 | -5.08% | [-5.74% , -4.36%] | - | - |
| Gavalas [generated] | - | 300.71 | 331.26 | -11.04% | [-12.09% , -10.04%] | - | - |
| Solomon [benchmark] | 624.59 | 607.12 | 607.55 | -0.35% | [-1.05% , 0.29%] | 2.19% | [1.64% , 2.73%] |
| Cordeau [benchmark] | 498.50 | 458.60 | 482.75 | -4.77% | [-6.65% , -2.95%] | 2.91% | [1.91% , 4.01%] |
| Gavalas [benchmark] | 307.64 | 298.88 | 311.15 | -4.52% | [-6.78% , -2.54%] | -0.82% | [-1.56% , -0.09%] |

Table 5.4: Average performance on all instances of a global model trained on all instance-regions and then fine-tuned during 50,000 epochs for each instance-region. Inference using beam search with a maximum number of 128 beams. The generated instances include n=64 generated instances for each of template benchmark instance-regions. The 95% confidence intervals are computed using bootstrap (sampling instance-regions with replacement). For detailed performance on each individual benchmark instance and average performance for each instance-region see Tables B.3, B.4, B.5, B.6, B.7 and B.8 in Supplementary materials.

6 Discussion

Here we show that a Pointer Network model trained using reinforcement learning is able to solve the Orienteering Problem with Time Windows. We approach the problem with the Tourist Trip Design Problem in mind, making sure that each model generalizes across different tourists who might visit the particular region(s) the model is trained on. We test our approach in both established and relatively new benchmark instances and show that it significantly outperforms the standard competitive heuristic ILS with inference times that are suitable for real time on-line applications. The first application of PNNs for solving np-hard combinatorial optimization problems such as the Travelling Salesman Problem and variations of the Vehicle Routing Problem, (Deudon et al. 2018; Nazari et al. 2018; Kool et al. 2019a). We customized the PN architecture specifically for the OPTW. This customization happens mainly on the encoding of the points of interest, which is significantly different from previously proposed PNNs models.

To start with, we use blocks of transformers (Deudon et al. 2018) that being permutation invariant, unlike RNNs (Nazari et al. 2018), are well suited for set encoding. Furthermore, we apply set encoding in a dynamic or iterative way. In particular, we use dynamical features that are updated at each iteration and we use one step lookahead to construct a graph representation of the admissible points of interest and use its adjacency matrix for self-attention masking. Importantly, we introduce a change in the transformer architecture: we add recursion by making the attention sub-block dependent on the previous iteration step. While these changes seem to improve the model's performance in this and other applications of the PNNs.

We have shown that we can get production-level performance and inference times. While ultimately the inference time is more important, the long training time can be inconvenient. Our main results are for a model trained for 500,000 epochs, which might take up to 72 hours. We show that training for 10% of the epochs produces higher scores than the ILS algorithm and we have explored several ways of improving training speed and performance. In particular, we show that having a better generative model of tourists or a better sampling strategy improves training, and we also show, using different practical training scenarios, that using warmed-up models, transfer learning and fine-tuning works and speeds-up training.

The possibility of having access to real data, e.g., tours and tourist feedback, could lead to models that enhance route usability and could better address a tourist’s preferences. The formal optimal solution of the abstract optimization problem may not always be the best practical solution in a real-world setting. The ideal tour may depend on many other features not considered in the OPTW, which could be implicitly addressed in a machine learning approach that is capable of leveraging real data. The exploration strategy in a reinforcement learning approach can also have a big impact on performance and learning speed. While we use beam search for inference, we use stochastic sampling as an exploration strategy. We briefly explored using stochastic beam search but obtained worse results. It would be interesting to try different exploration strategies, like for instance a Monte Carlo tree search (Browne et al. 2012). Another aspect of our exploration strategy is that

17
each batch has samples from the same generated instance or tourist. It would be interesting to see if the inclusion of more tourists in the same batch (i.e. $k > 1$ in Kool et al. (2019b)) can improve learning.

An advantage of a machine learning approach is that the output is probabilistic. This allows us, for instance, to retrieve ranked top-n solutions and use them in a broader route recommendation system. It also allows different inference strategies and the selection of the inference times according to the execution time requirements for the application at hand: we can make it faster by using almost instantaneous inference strategies like stochastic sampling or greedy inference, or by reducing the number of beams in beam search, or slower, but with better performance, by adding active search. This is an advantage over the several algorithmic approaches to the OPTW (Vansteenwegen and Gunawan (2019)) that often have execution times that make it difficult to include them in real-time applications.

Our approach is broad and flexible enough to be applied to different OPTW applications, like vehicle routing, transportation, scheduling, telecommunication, logistics and/or to other combinatorial optimization problems. The overall model architecture would remain the same. It would require the rewriting of admissibility conditions for node masking and lookahead search that are specific to the new optimization problem. It would also be necessary to choose features that make sense and are relevant for that specific problem. Furthermore, even for different OPTW applications the distribution from which training instances are sampled should reflect the specific application of interest. Altogether, the flexibility of the approach makes it a potentially useful tool for solving practical problems.

The source code can be found in GitHub[1].

Acknowledgements

The authors are grateful to Nishan Mann, Bahram Marami, João Gama and Hugo Penedones for their valuable suggestions. The first author is grateful to the Research Centre in Digital Services (CISeD), the Polytechnic of Viseu and FCT - Foundation for Science and Technology, I.P., within the scope of the project Refª UIDB/05583/2020 for their support.

References

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. 3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings, abs/1409.0473.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2017). Neural Combinatorial Optimization with Reinforcement Learning. Proceedings of the 5th International Conference on Learning Representations (ICLR).

Bengio, Y., Lodi, A., and Prouvost, A. (2020). Machine learning for combinatorial optimization: A methodological tour d’horizon. European Journal of Operational Research.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games, 4(1):1–43.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. Advances in Neural Information Processing Systems, 2017-December:6349–6359.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., and Rousseau, L.-M. (2018). Learning Heuristics for the TSP by Policy Gradient. In Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR), pages 170–181.

Falkner, J. K. and Schmidt-Thieme, L. (2020). Learning to solve vehicle routing problems with time windows through joint attention. arXiv: 2006.091005.

Gavalas, D., Konstantopoulos, C., Mastakas, K., and Pantziou, G. (2014). A survey on algorithmic approaches for solving tourist trip design problems. Journal of Heuristics, 20(3):291–328.

Gavalas, D., Konstantopoulos, C., Mastakas, K., and Pantziou, G. (2019). Efficient Cluster-Based Heuristics for the Team Orienteering Problem with Time Windows. Asia-Pacific Journal of Operational Research, 36(01):1–44.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. Journal of Machine Learning Research, 9:249–256.

Gu, S. and Hao, T. (2018). A Pointer Network Based Deep Learning Algorithm for 0-1 Knapsack Problem. 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI), pages 473–477.

[1]https://github.com/mustelideos/optw_rl
Gu, S. and Yang, Y. (2018). A Pointer Network Based Deep Learning Algorithm for the Max-Cut Problem. *Neural Information Processing. ICONIP 2018. Lecture Notes in Computer Science*, 11301:238–248.

Gunawan, A., Hoong Chuin, L., and LKun, L. (2015). *An Iterated Local Search Algorithm for Solving the Orienteering Problem with Time Windows*, volume 9026 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham.

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

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Karabulut, K. and Tasgetiren, M. F. (2020). An evolution strategy approach to the team orienteering problem with time windows. *Computers & Industrial Engineering*, 139:106109.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, abs/1412.6980.

Kool, W., Van Hoof, H., and Welling, M. (2019a). Attention, learn to solve routing problems! *7th International Conference on Learning Representations, ICLR 2019*, pages 1–25.

Kool, W., Van Hoof, H., and Welling, M. (2019b). Buy 4 reinforce samples, get a baseline for free! *Deep Reinforcement Learning Meets Structured Prediction, DeepRLStructPred@ICLR 2019 Workshop*, pages 1–14.

Lin, B., Ghaddar, B., and Nathwani, J. (2020). Deep reinforcement learning for electric vehicle routing problem with time windows. arXiv: 2010.020685.

Nazari, M., Oroojlooy, A., Snyder, L. V., and Takáč, M. (2018). Deep Reinforcement Learning for Solving the Vehicle Routing Problem. In *Proceedings Neural Information Processing Systems (NIPS)*, pages 9839–9849.

Schmid, V. and Ehmke, J. F. (2017). An Effective Large Neighborhood Search for the Team Orienteering Problem with Time Windows. *Computational Logistics. ICCL. Lecture Notes in Computer Science*, 10572:3–18.

Vansteenwegen, P. and Gunawan, A. (2019). *Orienteering Problems, Models and Algorithms for Vehicle Routing Problems with Profits*. Springer, euro advan edition.

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

Vansteenwegen, P., Souffriau, W., Vanden Berghe, G., and Van Oudheusden, D. (2009). Iterated local search for the team orienteering problem with time windows. *Computers & Operations Research*, 36(12):3281–3290.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need. *NIPS’17: Proceedings of the 31st International Conference on Neural Information Processing Systems*, page 6000–6010.

Veličković, P., Casanova, A., Liò, P., Cucurull, G., Romero, A., and Bengio, Y. (2018). Graph attention networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pages 1–12.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256.
A Model Comparison: Recursion in the Transformer and Graph Masked Self-Attention

We introduce some changes to the Pointer Neural Network used in previous applications to combinatorial optimization problems (see Section 3). In this section we investigate the impact of two of those changes. Specifically, we compare our model to the exact same model but without recursion in the transformer (see Section 3.1.2 and "no recursion" in Figures A.1, A.2 and A.3) and to the exact same model but without the look-ahead graph masked self-attention (see Section 3.1.3 and "complete graph" in Figures A.1, A.2 and A.3). For this model comparison we use the global model training scheme without fine-tuning (see Section 4.6), i.e., we train a model for each group of instances on all instances of each group at the same time: in each epoch, a tourist-instance is generated for/from a benchmark instance-region template sampled randomly from all instances of that group. We find that our model outperforms the models that lack each of those two changes. The difference can be observed during training (using greedy inference) (Fig. A.1) and after training for beam search inference (Figs. A.2 and A.3) on both the benchmark and generated instances.

Figure A.1: Evolution of model performance (gap to best-known) during training using greedy inference, for the three models: proposed model (red), without recursion (blue), and without taking into account the look-ahead graph structure for masking (purple). The average model score gap and ILS-score gap to best-known are computed over all benchmark instances of the correspondent group (n=56 for Solomon, n=20 for Cordeau and n=33 for Gavalas). The shaded area represents 90% confidence intervals (bootstrap benchmark instances).

Figure A.2: Evolution of model performance (gap to best-known) using beam search for inference with up to a maximum number of 128 beams, for the three models: proposed model (red), without recursion (blue), and without taking into account the look-ahead graph structure for masking (purple). The average score gap and ILS-score gap to the best-known scores are computed over all benchmark instances of the correspondent group (n=56 for Solomon, n=20 for Cordeau and n=33 for Gavalas). The shaded area represents 90% confidence intervals (bootstrap benchmark instances).
Figure A.3: Performance (gap to ILS) using beam search for inference with up to a maximum number of 128 beams, for the three models: proposed model (red), without recursion (blue), and without taking into account the look-ahead graph structure for masking (purple). The average score gap to the ILS score is computed over the mean score of all generated instances (n=64 generated instances per instance-region). The (very-)shaded area represents 90% confidence intervals (bootstrap instance-regions).
## B Detailed Model Performance for All Instances

### B.1 Model Trained from Scratch

Here we present the detailed individual results for the subset of 28 instances (n=12 for Solomon, n=8 for Cordeau and n=8 for Gavalas, see Section 4 Methods for details). We evaluate performance on both benchmark (Table B.1) and generated (Table B.2) instances for the model(s) trained from scratch for 500,000 epochs and using beam search ("model") inference or active search followed by beam search ("model+as").

#### Benchmark Instances

| group | instance | best-known (bk) | ILS | model | model gap to ILS | model gap to bk | model+as | model+as gap to ILS | model+as gap to bk |
|-------|----------|-----------------|-----|-------|-----------------|---------------|----------|---------------------|------------------|
| Solomon | r101 | 198 | 182 | 198 | -8.79% | 0.00% | 198 | -8.79% | 0.00% |
| | r102 | 286 | 286 | 286 | 0.00% | 0.00% | 286 | 0.00% | 0.00% |
| | r201 | 797 | 788 | 794 | -0.63% | 0.50% | 793 | -0.76% | 0.38% |
| | r202 | 930 | 880 | 890 | -1.14% | 1.34% | 890 | -1.14% | 1.34% |
| | r101 | 219 | 219 | 219 | 0.00% | 0.00% | 219 | 0.00% | 0.00% |
| | r102 | 265 | 259 | 266 | -2.70% | 0.00% | 266 | -2.70% | 0.00% |
| | r201 | 795 | 780 | 791 | -1.79% | 0.13% | 794 | -1.79% | 0.13% |
| | r202 | 836 | 832 | 855 | -0.61% | 0.21% | 835 | -0.61% | 0.21% |
| | c101 | 320 | 320 | 320 | 0.00% | 0.00% | 320 | 0.00% | 0.00% |
| | c102 | 360 | 360 | 360 | 0.00% | 0.00% | 360 | 0.00% | 0.00% |
| | c201 | 870 | 840 | 870 | -3.57% | 0.00% | 870 | -3.57% | 0.00% |
| | c202 | 930 | 890 | 930 | -2.20% | 0.00% | 930 | -2.20% | 0.00% |

Table B.1: Performance (score, gap to ILS and gap to best-known) for models trained from scratch for 500,000 epochs using beam-search for inference ("model") or 128 epochs of active search before beam-search ("model+as"), for each of the benchmark instances in the subset of instances (12 Solomon, 8 Cordeau and 8 Gavalas).
Generated Instances

| Group | Instance [generated] | ILS | model gap to ILS | model+as | model+as gap to ILS |
|-------|----------------------|-----|------------------|----------|------------------|
| Solomon | r101 | 108.34 | 115.45 | -6.56% | 115.45 | -6.56% |
| | r102 | 135.72 | 166.19 | -18.88% | 166.19 | -18.88% |
| | r201 | 790.81 | 800.88 | -1.30% | 800.88 | -1.30% |
| | r102 | 152.08 | 159.06 | -4.47% | 159.06 | -4.47% |
| | r201 | 179.44 | 185.72 | -3.06% | 185.72 | -3.06% |
| | r202 | 517.08 | 527.36 | -2.00% | 527.36 | -2.00% |
| | r102 | 790.81 | 800.88 | -1.30% | 800.88 | -1.30% |
| | r201 | 235.92 | 265.61 | -12.19% | 265.61 | -12.19% |
| | r202 | 200.80 | 290.00 | -30.00% | 290.00 | -30.00% |
| | c101 | 255.92 | 265.61 | -3.79% | 265.61 | -3.79% |
| | c102 | 290.80 | 298.00 | -2.48% | 298.00 | -2.48% |
| | c201 | 687.45 | 693.98 | -0.95% | 693.98 | -0.95% |
| | c202 | 731.44 | 762.25 | -4.13% | 762.25 | -4.13% |
| Cordeau | pr01 | 191.72 | 199.64 | -0.83% | 199.64 | -0.83% |
| | pr02 | 204.42 | 213.53 | -4.56% | 213.53 | -4.56% |
| | pr03 | 246.08 | 263.34 | -6.72% | 263.34 | -6.72% |
| | pr04 | 304.45 | 323.97 | -5.79% | 323.97 | -5.79% |
| | pr11 | 268.51 | 285.05 | -6.34% | 285.05 | -6.34% |
| | pr12 | 335.00 | 359.44 | -6.94% | 359.44 | -6.94% |
| | pr13 | 294.72 | 304.38 | -3.32% | 304.38 | -3.32% |
| | pr14 | 346.60 | 372.55 | -7.05% | 372.55 | -7.05% |
| Gavalas | t101 | 320.44 | 339.16 | -5.84% | 339.16 | -5.84% |
| | t105 | 363.44 | 353.23 | -8.31% | 353.23 | -8.31% |
| | t14 | 328.36 | 358.55 | -8.70% | 358.55 | -8.70% |
| | t17 | 401.27 | 457.23 | -13.95% | 457.23 | -13.95% |
| | t201 | 235.88 | 266.36 | -12.92% | 266.36 | -12.92% |
| | t202 | 252.59 | 302.53 | -19.77% | 302.53 | -19.77% |
| | t203 | 250.79 | 273.94 | -8.84% | 273.94 | -8.84% |
| | t204 | 247.53 | 270.39 | -12.87% | 270.39 | -12.87% |

Table B.2: Average performance (score and gap to ILS) for models trained from scratch for 500,000 epochs using beam-search for inference ("model") or 128 epochs of active search before beam-search ("model+as"), for each of the generated instance-regions in the subset of instances (12 Solomon, 8 Cordeau and 8 Gavalas). The values are averages across the n=64 generated instances for each of template benchmark instance-regions.

B.2 Fine-tuned Global Model

We want to present results for all the instance-regions of each group of benchmark instances without having to go through the time-consuming process of training from scratch each and every one of them. In order to achieve that goal, we train a global model for each instance group i.e., a model trained simultaneously on all the instance-regions of that group, and then fine-tune it for 10% (50,000) of the epochs on each instance-region (see “Fine-tune a global model for every instance” in Section 5).
### Table B.3: Performance (score, gap to ILS and gap to best-known) on Solomon benchmark instances of a global model trained on all instance-regions of this group and then fine-tuned during 50,000 epochs for each instance-region ("global+ft"). Inference uses beam search with a maximum of 128 beams.
Table B.4: Average performance (score and gap to ILS) on Solomon generated instances of a global model trained on all instance-regions of this group and then fine-tuned during 50,000 epochs for each instance-region (“global+ft”). Inference uses beam search with a maximum of 128 beams. The values are averages across the n=64 generated instances.
Cordeau [Benchmark]

| instance | best-known | ILS | global-ft | gap to ILS | gap to best-known | instance | best-known | ILS | global-ft | gap to ILS | gap to best-known |
|----------|------------|-----|-----------|------------|-------------------|----------|------------|-----|-----------|------------|-------------------|
| pr01     | 308        | 304 | 308       | -1.32%     | 0.00%             | pr11     | 353        | 330 | 351       | -6.96%     | 0.37%             |
| pr02     | 404        | 385 | 405       | -4.15%     | 0.74%             | pr12     | 442        | 431 | 431       | -1.18%     | 0.36%             |
| pr03     | 394        | 384 | 390       | 1.34%      | 3.35%             | pr13     | 467        | 450 | 453       | -0.67%     | 0.00%             |
| pr04     | 489        | 447 | 456       | -6.49%     | 3.66%             | pr14     | 567        | 482 | 536       | -11.20%    | 5.47%             |
| pr05     | 505        | 576 | 591       | -2.60%     | 0.17%             | pr15     | 708        | 638 | 699       | -9.56%     | 1.27%             |
| pr06     | 591        | 538 | 567       | -5.39%     | 4.06%             | pr16     | 674        | 599 | 699       | -8.54%     | 9.64%             |
| pr07     | 298        | 291 | 293       | 0.00%      | 1.68%             | pr17     | 562        | 546 | 549       | -0.57%     | 3.59%             |
| pr08     | 463        | 463 | 461       | 0.43%      | 0.43%             | pr18     | 539        | 479 | 532       | -11.06%    | 1.30%             |
| pr09     | 493        | 461 | 466       | -1.01%     | 3.48%             | pr19     | 562        | 499 | 522       | -4.61%     | 7.12%             |
| pr10     | 504        | 539 | 580       | -7.61%     | 2.36%             | pr20     | 667        | 570 | 645       | -13.06%    | 3.30%             |

Table B.5: Performance (score, gap to ILS and gap to best-known) on Cordeau instances of a global model trained on all instance-regions of this group and then fine-tuned during 50,000 epochs for each instance-region ("global+ft"). Inference uses beam search with a maximum of 128 beams.

Cordeau [Generated]

| instance [generated] | ILS | global-ft | gap to ILS | instance [generated] | ILS | global-ft | gap to ILS |
|----------------------|-----|-----------|------------|----------------------|-----|-----------|------------|
| pr01                 | 191.72 | 199.77 | -4.20%     | pr11                 | 208.31 | 215.77 | -3.56%     |
| pr02                 | 208.42 | 215.78 | -4.58%     | pr12                 | 233.09 | 238.17 | -2.18%     |
| pr03                 | 246.08 | 261.67 | -6.34%     | pr13                 | 294.72 | 304.12 | -3.19%     |
| pr04                 | 304.45 | 321.86 | -5.72%     | pr14                 | 346.69 | 369.47 | -6.63%     |
| pr05                 | 291.12 | 310.86 | -6.70%     | pr15                 | 332.14 | 354.41 | -6.76%     |
| pr06                 | 325.52 | 342.89 | -5.54%     | pr16                 | 363.42 | 391.25 | -7.66%     |
| pr07                 | 174.28 | 181.53 | -4.16%     | pr17                 | 208.69 | 212.42 | -1.76%     |
| pr08                 | 246.45 | 256.48 | -4.07%     | pr18                 | 274.47 | 286.94 | -4.51%     |
| pr09                 | 269.58 | 307.12 | -4.06%     | pr19                 | 325.72 | 345.70 | -6.14%     |
| pr10                 | 341.28 | 361.90 | -5.92%     | pr20                 | 381.05 | 403.94 | -6.05%     |

Table B.6: Average performance (score and gap to ILS) on Cordeau generated instances of a global model trained on all instance-regions of this group and then fine-tuned during 50,000 epochs for each instance-region ("global+ft"). Inference uses beam search with a maximum of 128 beams. The values are averages across the n=64 generated instances.
Table B.7: Performance (score, gap to ILS and gap to best-known) on Gavalas instances of a global model trained on all instance-regions of this group and then fine-tuned during 50,000 epochs for each instance-region ("global+ft"). Inference uses beam search with a maximum of 128 beams.

| instance [generated] | ILS | global+ft | global+ft gap to ILS | instance [generated] | ILS | global+ft | global+ft gap to ILS |
|----------------------|-----|-----------|----------------------|----------------------|-----|-----------|----------------------|
| t101                 | 520.44 | 330.11    | -39.6%               | t130                 | 259.88 | 263.30    | -13.0%               |
| t108                 | 373.58 | 400.97    | -8.3%                | t131                 | 304.47 | 287.25    | -15.9%               |
| t114                 | 428.56 | 356.80    | -9.18%               | t132                 | 352.53 | 357.88    | -5.6%                |
| t117                 | 401.27 | 455.58    | -13.54%              | t121                 | 306.33 | 360.17    | -12.62%              |
| t122                 | 357.58 | 400.97    | -13.53%              | t119                 | 326.97 | 360.83    | -10.36%              |
| t123                 | 301.97 | 330.45    | -12.08%              | t131                 | 352.12 | 351.38    | -0.41%               |
| t132                 | 352.53 | 357.88    | -5.6%                | t133                 | 351.36 | 394.22    | -12.20%              |
| t143                 | 396.69 | 416.88    | -10.91%              | t145                 | 320.66 | 349.05    | -8.5%                |
| t148                 | 342.66 | 372.66    | -8.76%               | t150                 | 394.69 | 452.02    | -14.96%              |
| -                    | -     | -         | -                    | t150                 | 394.69 | 452.02    | -14.96%              |

Table B.8: Average performance (score and gap to ILS) on Gavalas generated instances of a global model trained on all instance-regions of this group and then fine-tuned during 50,000 epochs for each instance-region ("global+ft"). Inference uses beam search with a maximum of 128 beams. The values are averages across the n=64 generated instances.

| instance [generated] | ILS | global+ft | global+ft gap to ILS | instance [generated] | ILS | global+ft | global+ft gap to ILS |
|----------------------|-----|-----------|----------------------|----------------------|-----|-----------|----------------------|
| t101                 | 387 | 387       | -1.88%               | t105                 | 433 | 428       | 1.15%               |
| t114                 | 476 | 467       | -0.57%               | t117                 | 462 | 482       | -6.64%              |
| t121                 | 436 | 424       | -6.13%               | t122                 | 478 | 468       | 1.90%               |
| t123                 | 409 | 484       | -4.46%               | t124                 | 471 | 456       | -7.59%              |
| t126                 | 415 | 413       | -4.60%               | t128                 | 441 | 440       | -3.25%              |
| t131                 | 413 | 401       | -3.25%               | t132                 | 420 | 440       | -4.76%              |
| t134                 | 417 | 413       | -1.45%               | t136                 | 346 | 390       | -12.66%             |
| t145                 | 371 | 387       | -4.41%               | t146                 | 371 | 351       | -8.05%              |
| t148                 | 471 | 468       | -1.50%               | t150                 | 487 | 485       | 0.41%               |

C Benchmark Instances Data

In this section we present visualizations that should give information about the characteristics of each benchmark instance as well as the variability within and across each instance group.
C.1 Spacial Distribution

Solomon

Figure C.1: Spacial distribution of the points of interest (red), and starting and ending location (black) for the benchmark instances in the Solomon group.

Cordeau

Figure C.2: Spacial distribution of the points of interest (red), and starting and ending location (black) for the benchmark instances in the Cordeau group.
Gavalas

Figure C.3: Spacial distribution of the points of interest (red), and starting and ending location (black) for the benchmark instances in the Gavalas group.

C.2 Correlation between Scores and Duration of Visit

Solomon

Figure C.4: Score of each point of interest as a function of the time it takes to visit that point of interest for each benchmark instance in the Solomon group.
Cordeau

Figure C.5: Score of each point of interest as a function of the time it takes to visit that point of interest for each benchmark instance in the Cordeau group.

Gavalas

Figure C.6: Score of each point of interest as a function of the time it takes to visit that point of interest for each benchmark instance in the Gavalas group.
C.3 Schedules

Solomon

Figure C.7: Schedule (opening time to closing time) of each point of interest for all benchmark instances of the Solomon group.
Cordeau

Figure C.8: Schedule (opening time to closing time) of each point of interest for all benchmark instances of the Cordeau group.

Gavalas

Figure C.9: Schedule (opening time to closing time) of each point of interest for all benchmark instances of the Gavalas group.