# Deep Policy Dynamic Programming for Vehicle Routing Problems

Kool, W.; van Hoof, H.; Gromicho, J.; Welling, M.

DOI
10.48550/arXiv.2102.11756
10.1007/978-3-031-08011-1_14

Publication date
2022

Document Version
Submitted manuscript

Published in
Integration of Constraint Programming, Artificial Intelligence, and Operations Research

# Deep Policy Dynamic Programming for Vehicle Routing Problems

Wouter Kool*[1,2], Herke van Hoof[1], Joaquim Gromicho[1,2] and Max Welling[1]

[1]University of Amsterdam
[2]ORTEC

**Abstract**

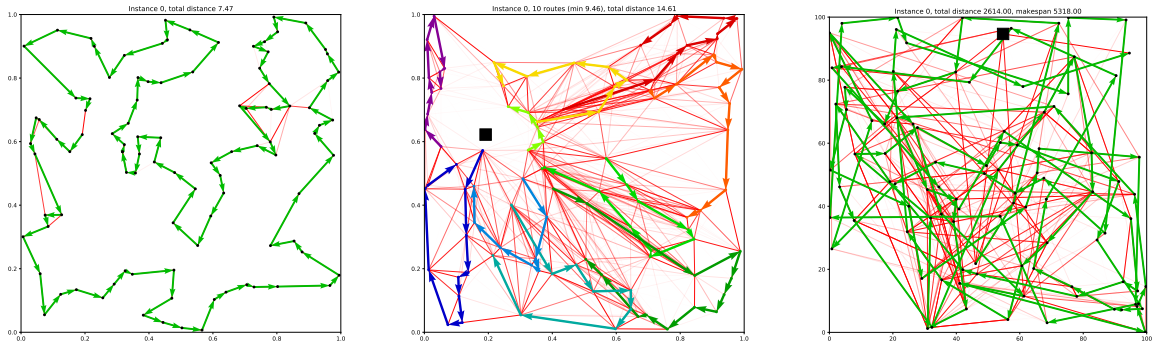
Routing problems are a class of combinatorial problems with many practical applications. Recently, end-to-end deep learning methods have been proposed to learn approximate solution heuristics for such problems. In contrast, classical dynamic programming (DP) algorithms guarantee optimal solutions, but scale badly with the problem size. We propose *Deep Policy Dynamic Programming* (DPDP), which aims to combine the strengths of learned neural heuristics with those of DP algorithms. DPDP prioritizes and restricts the DP state space using a policy derived from a deep neural network, which is trained to predict edges from example solutions. We evaluate our framework on the travelling salesman problem (TSP), the vehicle routing problem (VRP) and TSP with time windows (TSPTW) and show that the neural policy improves the performance of (restricted) DP algorithms, making them competitive to strong alternatives such as LKH, while also outperforming most other 'neural approaches' for solving TSPs, VRPs and TSPTWs with 100 nodes.

## 1 Introduction

Dynamic programming (DP) is a powerful framework for solving optimization problems by solving smaller subproblems through the principle of optimality [3]. Famous examples are Dijkstra's algorithm [14] for the shortest route between two locations, and the classic Held-Karp algorithm for the travelling salesman problem (TSP) [23, 4]. Despite their long history, dynamic programming algorithms for vehicle routing problems (VRPs) have seen limited use in practice, primarily due to their bad scaling performance. More recently, a line of research has attempted the use of machine learning (especially deep learning) to automatically learn heuristics for solving routing problems [61, 5, 45, 31, 7]. While the results are promising, most learned heuristics are not (yet) competitive to 'traditional' algorithms such as LKH [24] and lack (asymptotic) guarantees on their performance.

In this paper, we propose *Deep Policy Dynamic Programming* (DPDP) as a framework for solving vehicle routing problems. The key of DPDP is to combine the strengths of deep learning and DP, by restricting the DP state space (the search space) using a policy derived from a neural network. In Figure 1 it can be seen how the neural network indicates promising parts of the search space (through a *heatmap* over the edges of the graph), which is then used by the DP algorithm to find a good solution. DPDP is more powerful than some related ideas [67, 56, 66, 6, 37] as it combines

*Corresponding author: w.w.m.kool@uva.nl.




(a) Travelling Salesman Problem (b) Vehicle Routing Problem (c) TSP with Time Windows

Figure 1: Heatmap predictions (red) and solutions (colored) by DPDP (VRP depot edges omitted).

supervised training of a large neural network with just a *single* model evaluation at test time, to enable running a large scale guided search using DP. The DP framework is flexible as it can model a variety of realistic routing problems with difficult practical constraints [20]. We illustrate this by testing DPDP on the TSP, the capacitated VRP and the TSP with (hard) time window constraints (TSPTW).

In more detail, the starting point of our proposed approach is a *restricted dynamic programming* algorithm [20]. Such an algorithm heuristically reduces the search space by retaining only the $B$ most promising solutions per iteration. The selection process is very important as it defines the part of the DP state space considered and, thus, the quality of the solution found (see Fig. 2). Instead of manually defining a selection criterion, DPDP defines it using a (sparse) heatmap of promising route segments obtained by pre-processing the problem instance using a (deep) graph neural network (GNN) [27]. This approach is reminiscent of neural branching policies for branch-and-bound algorithms [19, 44].

In this work, we thus aim for a 'neural boost' of DP algorithms, by using a GNN for scoring partial solutions. Prior work on 'neural' vehicle routing has focused on auto-regressive models [61, 5, 13, 31], but they have high computational cost when combined with (any form of) search, as the model needs to be evaluated for each partial solution considered. Instead, we use a model to predict a heatmap indicating promising edges [27], and define the *score* of a partial solution as the 'heat' of the edges it contains (plus an estimate of the 'heat-to-go' or *potential* of the solution). As the neural network only needs to be evaluated *once* for each instance, this enables a *much larger search* (defined by $B$), making a good trade-off between quality and computational cost. Additionally, we can apply a threshold to the heatmap to define a sparse graph on which to run the DP algorithm, reducing the runtime by eliminating many solutions.

Figure 2 illustrates DPDP. In Section 4, we show that DPDP significantly improves over 'classic' restricted DP algorithms. Additionally, we show that DPDP outperformes most other 'neural' approaches for TSP, VRP and TSPTW and is competitive with the highly-optimized LKH solver [24] for VRP, while achieving similar results much faster for TSP and TSPTW. For TSPTW, DPDP also outperforms the best open-source solver we could find [10], illustrating the power of DPDP to handle difficult hard constraints (time windows).

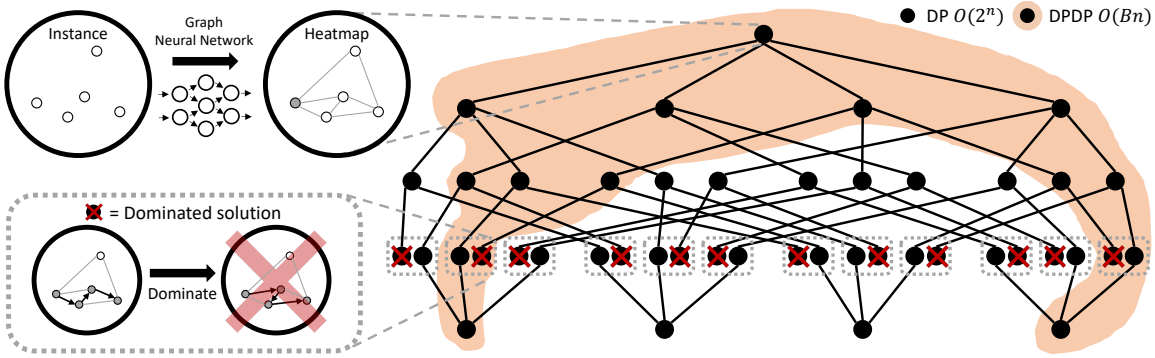



Figure 2: DPDP for the TSP. A GNN creates a (sparse) heatmap indicating promising edges, after which a tour is constructed using forward dynamic programming. In each step, at most $B$ solutions are expanded according to the heatmap policy, restricting the size of the search space. Partial solutions are dominated by shorter (lower cost) solutions with the same DP state: the same nodes visited (marked grey) and current node (indicated by dashed rectangles).

## 2 Related work

DP has a long history as an exact solution method for routing problems [33, 54], e.g. the TSP with time windows [15] and precedence constraints [43], but is limited to small problems due to the curse of dimensionality. Restricted DP (with heuristic policies) has been used to address, e.g., the time dependent TSP [41], and has been generalized into a flexible framework for VRPs with different types of practical constraints [20]. DP approaches have also been shown to be useful in settings with difficult practical issues such as time-dependent travel times and driving regulations [30] or stochastic demands [46]. For more examples of DP for routing (and scheduling), see [57]. For sparse graphs, alternative, but less flexible, formulations can be used [8].

Despite the flexibility, DP methods have not gained much popularity compared to heuristic search approaches such as R&R [51], ALNS [50], LKH [24], HGS [60, 59] or FILO [1], which, while effective, have limited flexibility as special operators are needed for different types of problems. While restricted DP was shown to have superior performance on *realistic* VRPs with many constraints [20], the performance gap of around 10% for standard (benchmark) VRPs (with time windows) is too large to popularize this approach. We argue that the missing ingredient is a strong but computationally cheap policy for selecting which solutions to considered, which is the motivation behind DPDP.

In the machine learning community, deep neural networks (DNNs) have recently boosted performance on various tasks [34]. After the first DNN model was trained (using example solutions) to construct TSP tours [61], many improvements have been proposed, e.g. different training strategies such as reinforcement learning (RL) [5, 28, 12, 32] and model architectures, which enabled the same idea to be used for other routing problems [45, 31, 13, 49, 16, 64, 40]. Most constructive neural methods are *auto-regressive*, evaluating the model many times to predict one node at the time, but other works have considered predicting a 'heatmap' of promising edges *at once* [47, 27, 17], which allows a tour to be constructed (using sampling or beam search) without further evaluating the model. An alternative to constructive methods is 'learning to search', where a neural network is used to guide a search procedure such as local search [7, 38, 18, 63, 26, 29, 36, 65, 25]. Scaling to instances beyond 100 nodes remains challenging [39, 17].

The combination of machine learning with DP has been proposed in limited settings [67, 56, 66].

Most related to our approach, a DP algorithm for TSPTW, guided by an RL agent, was implemented using an existing solver [6], which is less efficient than DPDP (see Section 4.3). Also similar to our approach, a neural network predicting edges has been combined with tree search [37] and local search for maximum independent set (MIS). Whereas DPDP directly builds on the idea of predicting promising edges [37, 27], it uses these more efficiently through a policy with *potential function* (see Section 3.1), and by using DP rather than tree search or beam search, we exploit known problem structure in a principled and general manner. As such, DPDP obtains strong performance without using extra heuristics such as local search. For a wider view on machine learning for routing problems and combinatorial optimization, see [42, 58].

# 3 Deep Policy Dynamic Programming

DPDP uses an existing graph neural network [27] (which we modify for VRP and TSPTW) to predict a heatmap of promising edges, which is used to derive the policy for scoring partial solutions in the DP algorithm. The DP algorithm starts with a *beam* of a single initial (empty) solution. It then proceeds by iterating the following steps: (1) all solutions on the beam are expanded, (2) dominated solutions are removed for each *DP state*, (3) the $B$ best solutions according to the scoring policy define the beam for the next iteration. These steps are illustrated in Fig. 2. The objective function is used to select the best solution from the final beam. The resulting algorithm is a *beam search* over the *DP state space* (which is *not* a 'standard beam search' over the *solution space*!), where $B$ is the *beam size*. DPDP is asymptotically optimal as using $B = n \cdot 2^n$ for a TSP with $n$ nodes guarantees optimal results, but choosing smaller $B$ allows to trade performance for computational cost.

DPDP is a generic framework that can be applied to different problems, by defining the following ingredients: (1) the **variables** to track while constructing solutions, (2) the **initial solution**, (3) **feasible actions** to expand solutions, (4) rules to define **dominated solutions** and (5) a **scoring policy** for selecting the $B$ solutions to keep. A solution is always (uniquely) defined as a sequence of actions, which allows the DP algorithm to construct the final solution by backtracking. In the next sections, we define these ingredients for the TSP, VRP and TSPTW.

## 3.1 Travelling Salesman Problem

We implement DPDP for Euclidean TSPs with $n$ nodes on a (sparse) graph, where the cost for edge $(i, j)$ is given by $c_{ij}$, the Euclidean distance between the nodes $i$ and $j$. The objective is to construct a tour that visits all nodes (and returns to the start node) and minimizes the total cost of its edges.

For each partial solution, defined by a sequence of actions $\boldsymbol{a}$, the **variables** we track are $\text{cost}(\boldsymbol{a})$, the total *cost* (distance), $\text{current}(\boldsymbol{a})$, the current node, and $\text{visited}(\boldsymbol{a})$, the set of visited nodes (including the start node). Without loss of generality, we let 0 be the start node, so we initialize the beam at step $t = 0$ with the empty **initial solution** with $\text{cost}(\boldsymbol{a}) = 0$, $\text{current}(\boldsymbol{a}) = 0$ and $\text{visited}(\boldsymbol{a}) = \{0\}$. At step $t$, the action $a_t \in \{0, ..., n-1\}$ indicates the next node to visit, and is a **feasible action** for a partial solution $\boldsymbol{a} = (a_0, ..., a_{t-1})$ if $(a_{t-1}, a_t)$ is an edge in the graph and $a_t \notin \text{visited}(\boldsymbol{a})$, or, when all are visited, if $a_t = 0$ to return to the start node. When expanding the solution to $\boldsymbol{a}' = (a_0, ..., a_t)$, we can compute the tracked variables incrementally as:

$$\text{cost}(\boldsymbol{a}') = \text{cost}(\boldsymbol{a}) + c_{\text{current}(a),a_t},\ \text{current}(\boldsymbol{a}') = a_t,\ \text{visited}(\boldsymbol{a}') = \text{visited}(\boldsymbol{a}) \cup \{a_t\}. \tag{1}$$

A (partial) solution $\boldsymbol{a}$ is a **dominated solution** if there exists a (dominating) solution $\boldsymbol{a}^*$ such that $\text{visited}(\boldsymbol{a}^*) = \text{visited}(\boldsymbol{a})$, $\text{current}(\boldsymbol{a}^*) = \text{current}(\boldsymbol{a})$ and $\text{cost}(\boldsymbol{a}^*) < \text{cost}(\boldsymbol{a})$. We refer to the tuple $(\text{visited}(\boldsymbol{a}), \text{current}(\boldsymbol{a}))$ as the *DP state*, so removing all dominated partial solutions, we keep

exactly one minimum-cost solution for each unique DP state[1]. Since a solution can only dominate other solutions with the same set of visited nodes, we only need to remove dominated solutions from sets of solutions with the same number of actions. Therefore, we can efficiently execute the DP algorithm in iterations, where at step $t$ all solutions have (after $t$ actions) $t+1$ visited nodes (including the start node), keeping the memory need at $O(B)$ states (with $B$ the beam size).

We define the **scoring policy** using a pretrained model [27], which takes as input node coordinates and edge distances to predict a raw 'heatmap' value $\hat{h}_{ij} \in (0, 1)$ for each edge $(i, j)$. The model was trained to predict optimal solutions, so $\hat{h}_{ij}$ can be seen as the probability that edge $(i, j)$ is in the optimal tour. We force the heatmap to be symmetric thus we define $h_{ij} = \max\{\hat{h}_{ij}, \hat{h}_{ji}\}$. The policy is defined using the heatmap values, in such a way to select the (partial) solutions with the largest total *heat*, while also taking into account the (heat) *potential* for the unvisited nodes. The policy thus selects the $B$ solutions which have the highest *score*, defined as $\text{score}(\boldsymbol{a}) = \text{heat}(\boldsymbol{a}) + \text{potential}(\boldsymbol{a})$, with $\text{heat}(\boldsymbol{a}) = \sum_{i=1}^{t-1} h_{a_{i-1},a_i}$, i.e. the sum of the heat of the edges, which can be computed incrementally when expanding a solution. The potential is added as an estimate of the 'heat-to-go' (similar to the heuristic in $A^*$ search) for the remaining nodes, and avoids the 'greedy pitfall' of selecting the best edges while skipping over nearby nodes, which would prevent good edges from being used later. It is defined as $\text{potential}(\boldsymbol{a}) = \text{potential}_0(\boldsymbol{a}) + \sum_{i \notin \text{visited}(\boldsymbol{a})} \text{potential}_i(\boldsymbol{a})$ with $\text{potential}_i(\boldsymbol{a}) = w_i \sum_{j \notin \text{visited}(\boldsymbol{a})} \frac{h_{ji}}{\sum_{k=0}^{n-1} h_{ki}}$, where $w_i$ is the node *potential weight* given by $w_i = (\max_j h_{ji}) \cdot (1 - 0.1(\frac{c_{i0}}{\max_j c_{j0}} - 0.5))$. By normalizing the heatmap values for incoming edges, the (remaining) potential for node $i$ is initially equal to $w_i$ but decreases as good edges become infeasible due to neighbours being visited. The node potential weight $w_i$ is equal to the maximum incoming edge heatmap value (an upper bound to the heat contributed by node $i$), which gets multiplied by a factor 0.95 to 1.05 to give a higher weight to nodes closer to the start node, which we found helps to encourage the algorithm to keep edges that enable to return to the start node. The overall heat + potential function identifies promising partial solutions and is computationally cheap.

## 3.2 Vehicle Routing Problem

For the VRP, we add a special depot node DEP to the graph. Node $i$ has a demand $d_i$, and the goal is to minimize the cost for a set of routes that visit all nodes. Each route must start and end at the depot, and the total demand of its nodes cannot exceed the vehicle capacity denoted by CAPACITY.

Additionally to the TSP **variables** $\text{cost}(\boldsymbol{a})$, $\text{current}(\boldsymbol{a})$ and $\text{visited}(\boldsymbol{a})$, we keep track of $\text{capacity}(\boldsymbol{a})$, which is the *remaining* capacity in the current route/vehicle. A solution starts at the depot, so we initialize the beam at step $t = 0$ with the empty **initial solution** with $\text{cost}(\boldsymbol{a}) = 0$, $\text{current}(\boldsymbol{a}) = \text{DEP}$, $\text{visited}(\boldsymbol{a}) = \emptyset$ and $\text{capacity}(\boldsymbol{a}) = \text{CAPACITY}$. For the VRP, we do not consider visiting the depot as a separate action. Instead, we define $2n$ actions, where $a_t \in \{0, ..., 2n-1\}$. The actions $0, ..., n-1$ indicate a *direct* move from the current node to node $a_t$, whereas the actions $n, ..., 2n-1$ indicate a move to node $a_t - n$ *via the depot*. **Feasible actions** are those that move to unvisited nodes via edges in the graph and obey the following constraints. For the first action $a_0$ there is no choice and we constrain (for convenience of implementation) $a_0 \in \{n, ..., 2n-1\}$. A direct move ($a_t < n$) is only feasible if $d_{a_t} \leq \text{capacity}(\boldsymbol{a})$ and updates the state similar to TSP but reduces remaining capacity by $d_{a_t}$. A move via the depot is always feasible (respecting the graph edges and assuming $d_i \leq \text{CAPACITY}\ \forall i$) as it resets the vehicle CAPACITY before subtracting demand, but incurs the 'via-depot cost' $c_{ij}^{\text{DEP}} = c_{i,\text{DEP}} + c_{\text{DEP},j}$. When all nodes are visited, we allow a special action to return to the depot. This somewhat unusual way of representing a CVRP solution has desirable properties

[1]If we have multiple partial solutions with the same state and cost, we can arbitrarily choose one to dominate the other(s), for example the one with the lowest index of the current node.

similar to the TSP formulation: at step $t$ we have exactly $t$ nodes visited, and we can run the DP in iterations, removing dominated solutions at each step $t$.

For VRP, a partial solution $\boldsymbol{a}$ is a **dominated solution** dominated by $\boldsymbol{a}^*$ if visited($\boldsymbol{a}^*$) = visited($\boldsymbol{a}$) and current($\boldsymbol{a}^*$) = current($\boldsymbol{a}$) (i.e. $\boldsymbol{a}^*$ corresponds to the same DP state) and cost($\boldsymbol{a}^*$) $\leq$ cost($\boldsymbol{a}$) and capacity($\boldsymbol{a}^*$) $\geq$ capacity($\boldsymbol{a}$), with *at least one of the two inequalities being strict.* This means that for each DP state, given by the set of visited nodes and the current node, we do not only keep the (single) solution with lowest cost (as in the TSP algorithm), but keep the complete set of pareto-efficient solutions in terms of cost and remaining vehicle capacity. This is because a higher cost partial solution may still be preferred if it has more remaining vehicle capacity, and vice versa.

For the VRP **scoring policy**, we modify the model [27] to include the depot node and demands. The special depot node gets a separate learned initial embedding parameter, and we add additional edge types for connections to the depot, to mark the depot as being special. Additionally, each node gets an extra input (next to its coordinates) corresponding to $d_i/\text{CAPACITY}$ (where we set $d_{\text{DEP}} = 0$). Apart from this, the model remains exactly the same[2]. The model is trained on example solutions from LKH [24] (see Section 4.2), which are not optimal, but still provide a useful training signal. Compared to TSP, the definition of the heat is slightly changed to accommodate for the 'via-depot actions' and is best defined incrementally using the 'via-depot heat' $h_{ij}^{\text{DEP}} = h_{i,\text{DEP}} \cdot h_{\text{DEP},j} \cdot 0.1$, where multiplication is used to keep heat values interpretable as probabilities and in the range $(0, 1)$. The additional penalty factor of 0.1 for visiting the depot encourages the algorithm to minimize the number of vehicles/routes. The initial heat is 0 and when expanding a solution $\boldsymbol{a}$ to $\boldsymbol{a'}$ using action $a_t$, the heat is incremented with either $h_{\text{current}(\boldsymbol{a}),a_t}$ (if $a_t < n$) or $h^{\text{DEP}}_{\text{current}(\boldsymbol{a}),a_t-n}$ (if $a_t \geq n$). The potential is defined similarly to TSP, replacing the start node 0 by DEP.

### 3.3 Travelling Salesman Problem with Time Windows

For the TSPTW, we also have a special depot/start node 0. The goal is to create a single tour that visits each node $i$ in a time window defined by $(l_i, u_i)$, where the travel time from $i$ to $j$ is equal to the cost/distance $c_{ij}$. It is allowed to wait if arrival at node $i$ is before $l_i$, but arrival cannot be after $u_i$. We minimize the total *cost* (*excluding* waiting time), but to minimize *makespan* (including waiting time), we only need to train on different example solutions. Due to the hard constraints, TSPTW is typically considered more challenging than plain TSP, for which every solution is feasible.

The **variables** we track and **initial solution** are equal to TSP except that we add time($\boldsymbol{a}$) which is initially 0 ($= l_0$). **Feasible actions** $a_t \in \{0, ..., n-1\}$ are those that move to unvisited nodes via edges in the graph such that the arrival time is no later than $u_{a_t}$ and do not directly eliminate the possibility to visit other nodes in time[3]. Expanding a solution $\boldsymbol{a}$ to $\boldsymbol{a'}$ using action $a_t$ updates the time as $\text{time}(\boldsymbol{a'}) = \max\{\text{time}(\boldsymbol{a}) + c_{\text{current}(a),a_t}, l_{a_t}\}$.

For each DP state, we keep all efficient solutions in terms of cost and time, so a partial solution $\boldsymbol{a}$ is a **dominated solution** dominated by $\boldsymbol{a}^*$ if $\boldsymbol{a}^*$ has the same DP state (visited($\boldsymbol{a}^*$) = visited($\boldsymbol{a}$) and current($\boldsymbol{a}^*$) = current($\boldsymbol{a}$)) and is strictly better in terms of cost and time, i.e. cost($\boldsymbol{a}^*$) $\leq$ cost($\boldsymbol{a}$) and time($\boldsymbol{a}^*$) $\leq$ time($\boldsymbol{a}$), with *at least one of the two inequalities being strict.*

The model [27] for the **scoring policy** is adapted to include the time windows $(l_i, u_i)$ as node features (in the same unit as coordinates and costs), and we use a special embedding for the depot similar to VRP. Due to the time dimension, a TSPTW solution is *directed*, and edge $(i, j)$ may be good whereas $(j, i)$ may be not, so we adapt the model to enable predictions $h_{ij} \neq h_{ji}$ (see Appendix B). We generated example training solutions using (heuristic) DP with a large beam size, which was faster than LKH. Given the heat predictions, the score (heat + potential) is exactly as for TSP.

[2]Except that we do not use the K-nearest neighbour feature [27] as it contains no additional information.

[3]E.g., arriving at node $i$ at $t = 10$ is not feasible if node $j$ has $u_j = 12$ and $c_{ij} = 3$.

### 3.4 Graph sparsity

As described, the DP algorithm can take into account a sparse graph to define feasible expansions. As our problems are defined on sets of nodes rather than graphs, the use of a sparse graph is an artificial design choice, which allows to significantly reduce the runtime but may sacrifice the possibility to find good or optimal tours. We propose two different strategies for defining the sparse graph on which to run the DP: thresholding the heatmap values $h_{ij}$ and using the K-nearest neighbour (KNN) graph. By default, we use a (low) heatmap threshold of $10^{-5}$, which rules out most of the edges as the model confidently predicts (close to) 0 for most edges. This is a secondary way to leverage the neural network (independent of the scoring policy), which can be seen as a form of learned *problem reduction* [53]. For symmetric problems (TSP and VRP), we add KNN edges in both directions. For the VRP, we additionally connect each node to the depot (and vice versa) to ensure feasibility.

### 3.5 Implementation & hyperparameters

We implement DPDP using PyTorch [48] to leverage GPU computation. For details, see Appendix A. Our code is publicly available.[4] DPDP has very few hyperparameters, but the heatmap threshold of $10^{-5}$ and details like the functional form of e.g. the scoring policy are 'educated guesses' or manually tuned on a few validation instances and can likely be improved. The runtime is influenced by implementation choices which were tuned on a few validation instances.

## 4 Experiments

### 4.1 Travelling Salesman Problem

In Table 1 we report our main results for DPDP with beam sizes of 10K (10 thousand) and 100K, for the TSP with 100 nodes on a commonly used test set of 10000 instances [31]. We report cost and *optimality gap* (see [31]) using Concorde [2], LKH [24] and Gurobi [22], as well as recent results of the strongest methods using neural networks ('neural approaches') from literature. Running times for solving 10000 instances *after training* should be taken as rough indications as some are on different machines, typically with 1 GPU or a many-core CPU (8 - 32). The costs indicated with * are not directly comparable due to slight dataset differences [17]. Times for generating heatmaps (if applicable) is reported separately (as the first term) from the running time for MCTS [17] or DP. DPDP achieves close to optimal results, strictly outperforming the neural baselines achieving better results in less time (except POMO [32], see Section 4.2).

### 4.2 Vehicle Routing Problem

For the VRP, we train the model using 1 million instances of 100 nodes, generated according to the distribution described by [45] and solved using one run of LKH [24]. We train using a batch size of 48 and a learning rate of $10^{-3}$ (selected as the result of manual trials to best use our GPUs), for (at most) 1500 epochs of 500 training steps (following [27]) from which we select the saved checkpoint with the lowest validation loss. We use the validation and test sets by [31].

Table 1 shows the results compared to a recent implementation of Hybrid Genetic Search (HGS)[5], a SOTA heuristic VRP solver [60, 59]. HGS is faster and improves around 0.5% over LKH, which is typically considered the baseline in related work. We present the results for LKH, as well as the

[4] `https://github.com/wouterkool/dpdp`
[5] `https://github.com/vidalt/HGS-CVRP`

Table 1: Mean cost, gap and *total time* to solve 10000 TSP/VRP test instances.

| Problem | TSP100 | | | VRP100 | | |
|---|---|---|---|---|---|---|
| Method | Cost | Gap | Time | Cost | Gap | Time |
| Concorde [2] | 7.765 | 0.000 % | 6m | | | |
| Hybrid Genetic Search [60, 59] | | | | 15.563 | 0.000 % | 6h11m |
| Gurobi [22] | 7.776 | 0.151 % | 31m | | | |
| LKH [24] | 7.765 | 0.000 % | 42m | 15.647 | 0.536 % | 12h57m |
| GNN Heatmap + Beam Search [27] | 7.87 | 1.39 % | 40m | | | |
| Learning 2-opt heuristics [9] | 7.83 | 0.87 % | 41m | | | |
| Merged GNN Heatmap + MCTS [17] | 7.764* | 0.04 % | 4m + 11m | | | |
| Attention Model + Sampling [31] | 7.94 | 2.26 % | 1h | 16.23 | 4.28 % | 2h |
| Step-wise Attention Model [64] | 8.01 | 3.20 % | 29s | 16.49 | 5.96 % | 39s |
| Attn. Model + Coll. Policies [29] | 7.81 | 0.54 % | 12h | 15.98 | 2.68 % | 5h |
| Learning improv. heuristics [63] | 7.87 | 1.42 % | 2h | 16.03 | 3.00 % | 5h |
| Dual-Aspect Coll. Transformer [40] | 7.77 | 0.09 % | 5h | 15.71 | 0.94 % | 9h |
| Attention Model + POMO [32] | 7.77 | 0.14 % | 1m | 15.76 | 1.26 % | 2m |
| NeuRewriter [7] | | | | 16.10 | 3.45 % | 1h |
| Dynamic Attn. Model + 2-opt [49] | | | | 16.27 | 4.54 % | 6h |
| Neur. Lrg. Neighb. Search [26] | | | | 15.99 | 2.74 % | 1h |
| Learn to improve [38] | | | | 15.57* | - | 4000h |
| DPDP 10K | 7.765 | 0.009 % | 10m + 16m | 15.830 | 1.713 % | 10m + 50m |
| DPDP 100K | 7.765 | 0.004 % | 10m + 2h35m | 15.694 | 0.843 % | 10m + 5h48m |
| DPDP 1M | | | | 15.627 | 0.409 % | 10m + 48h27m |

strongest neural approaches and DPDP with beam sizes up to 1 million. Some results used 2000 (different) instances [38] and cannot be directly compared[6]. DPDP outperforms all other neural baselines, except POMO [32], which delivers good results very quickly by exploiting symmetries in the problem. However, as it cannot (easily) improve further with additional runtime, we consider this contribution orthogonal to DPDP. DPDP is competitive to LKH (see also Section 4.4).

**More realistic instances** We also train the model and run experiments with instances with 100 nodes from a more realistic and challenging data distribution [55]. This distribution, commonly used in the routing community, has greater variability, in terms of node clustering and demand distributions. LKH failed to solve two of the test instances, which is because LKH by default uses a fixed number of routes equal to a lower bound, given by $\left\lceil \frac{\sum_{i=0}^{n-1} d_i}{\text{CAPACITY}} \right\rceil$, which may be infeasible[7]. Therefore we solve these instances by rerunning LKH with an unlimited number of allowed routes (which gives worse results, see Section 4.4).

DPDP was run on a machine with 4 GPUs, but we also report (estimated) runtimes for 1 GPU (1080Ti), and we compare against 16 or 32 CPUs for HGS and LKH. In Table 2 it can be seen that the difference with LKH is, as expected, slightly larger than for the simpler dataset, but still below 1% for beam sizes of 100K-1M. We also observed a higher validation loss, so it may be possible to improve results using more training data. Nevertheless, finding solutions within 1% of the specialized SOTA HGS algorithm, and even closer to LKH, is impressive for these challenging instances, and we consider the runtime (for solving 10K instances) acceptable, especially when using multiple GPUs.

[6]The running time of 4000 hours (167 days) is estimated from 24min/instance [38].

[7]For example, three nodes with a demand of two cannot be assigned to two routes with a capacity of three.

Table 2: Mean cost, gap and *total time* to solve 10000 realistic VRP100 instances.

| Method | Cost | Gap | Time (1 GPU or 16 CPUs) | Time (4 GPUs or 32 CPUs) |
|---|---|---|---|---|
| HGS [60, 59] | 18050 | 0.000 % | 7h53m | 3h56m |
| LKH [24] | 18133 | 0.507 % | 25h32m | 12h46m |
| DPDP 10K | 18414 | 2.018 % | 10m + 50m | 2m + 13m |
| DPDP 100K | 18253 | 1.127 % | 10m + 5h48m | 2m + 1h27m |
| DPDP 1M | 18168 | 0.659 % | 10m + 48h27m | 2m + 12h7m |

### 4.3 TSP with Time Windows

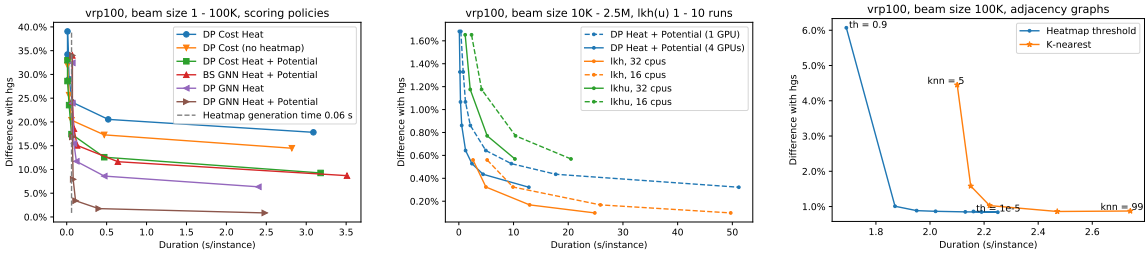
For the TSP with hard time window constraints, we use the data distribution by [6] and use their set of 100 test instances with 100 nodes. These were generated with small time windows, resulting in a small feasible search space, such that even with very small beam sizes, our DP implementation solves these instances optimally, eliminating the need for a policy. Therefore, we also consider a more difficult distribution similar to [10], which has larger time windows which are more difficult as the feasible search space is larger[8] [15]. For details, see Appendix B. For both distributions, we generate training data and train the model exactly as we did for the VRP.

Table 3 shows the results for both data distributions, which are reported in terms of the difference to General Variable Neighbourhood Search (GVNS) [10], the best open-source solver for TSPTW we could find[9], using 30 runs. For the small time window setting, both GVNS and DPDP find optimal solutions for all 100 instances in just 7 seconds (in total, either on 16 CPUs or a single GPU). LKH fails to solve one instance, but finds close to optimal solutions, but around 50 times slower. BaB-DQN* and ILDS-DQN* [6], methods combining an existing solver with an RL trained neural policy, take around 15 minutes *per instance* (orders of magnitudes slower) to solve most instances to optimality. Due to complex set-up, we were unable to run BaB-DQN* and ILDS-DQN* ourselves for the setting with larger time windows. In this setting, we find DPDP outperforms both LKH (where DPDP is orders of magnitude faster) and GVNS, in both speed and solution quality. This illustrates that DPDP, due to its nature, is especially well suited to handle constrained problems.

[8] Up to a limit, as making the time windows infinite size reduces the problem to plain TSP.
[9] `https://github.com/sashakh/TSPTW`

Table 3: Mean cost, gap and *total time* to solve TSPTW100 instances.

| Problem | Small time windows [6] (100 inst.) | | | | Large time windows [10] (10K inst.) | | | |
|---|---|---|---|---|---|---|---|---|
| Method | Cost | Gap | Fail | Time | Cost | Gap | Fail | Time |
| GVNS 30x [10] | 5129.58 | 0.000 % | | 7s | 2432.112 | 0.000 % | | 37m15s |
| GVNS 1x [10] | 5129.58 | 0.000 % | | <1s | 2457.974 | 1.063 % | | 1m4s |
| LKH 1x [24] | 5130.32 | 0.014 % | 1.00 % | 5m48s | 2431.404 | -0.029 % | | 34h58m |
| BaB-DQN* [6] | 5130.51 | 0.018 % | | 25h | | | | |
| ILDS-DQN* [6] | 5130.45 | 0.017 % | | 25h | | | | |
| DPDP 10K | 5129.58 | 0.000 % | | 6s + 1s | 2431.143 | -0.040 % | | 10m + 8m7s |
| DPDP 100K | 5129.58 | 0.000 % | | 6s + 1s | 2430.880 | - 0.051 % | | 10m + 1h16m |




(a) Different scoring policies, as well as 'pure' beam search, for beam sizes 1, 10, 100, 1000, 10K, 100K.

(b) Beam sizes 10K, 25K, 50K, 100K, 250K, 500K, 1M, 2.5M compared against LKH(U) with 1, 2, 5 and 10 runs.

(c) Sparsities with heatmap thresholds 0.9, 0.5, 0.2, 0.1, $10^{-2}$, $10^{-3}$, $10^{-4}$, $10^{-5}$ and knn = 5, 10, 20, 50, 99. Beam size 100K.

Figure 3: DPDP ablations on 100 validation instances of VRP with 100 nodes.

## 4.4 Ablations

**Scoring policy** To evaluate the value of different components of DPDP's **GNN Heat + Potential** scoring policy, we compare against other variants. **GNN Heat** is the version without the potential, whereas **Cost Heat + Potential** and **Cost Heat** are variants that use a 'heuristic' $\hat{h}_{ij} = \frac{c_{ij}}{\max_k c_{ik}}$ instead of the GNN. **Cost** directly uses the current cost of the solution, and can be seen as 'classic' restricted DP. Finally, **BS GNN Heat + Potential** uses beam search without dynamic programming, i.e. without removing dominated solutions. To evaluate only the scoring policy, each variant uses the fully connected graph (knn $= n-1$). Figure 3a shows the value of DPDP's potential function, although even without it results are still significantly better than 'classic' heuristic DP variants using cost-based scoring policies. Also, it is clear that using DP significantly improves over a standard beam search (by removing dominated solutions). Lastly, the figure illustrates how the time for generating the heatmap using the neural network, despite its significant value, only makes up a small portion of the total runtime.

**Beam size** DPDP allows to trade off the performance vs. the runtime using the beam size $B$ (and to some extent the graph sparsity, see Section 4.4). We illustrate this trade-off in Figure 3b, where we evaluate DPDP on 100 validation instances for VRP, with different beam sizes from 10K to 2.5M. We also report the trade-off curve for the LKH(U), which is the strongest baseline that can also solve different problems. We vary the runtime using 1, 2, 5 and 10 runs (returning the best solution). LKHU(nlimited) is the version which allows an unlimited number of routes (see Section 4.2). It is hard to compare GPU vs CPU, so we report (estimated) runtimes for different hardware, i.e. 1 or 4 GPUs (with 3 CPUs per GPU) and 16 or 32 CPUs. We report the difference (i.e. the gap) with HGS, analog to how results are reported in Table 1. We emphasize that in most related work (e.g. [31]), the strongest baseline considered is one run of LKH, so we compare against a much stronger baseline. Also, our goal is not to outperform HGS (which is SOTA and specific to VRP) or LKH, but to show DPDP has reasonable performance, while being a flexible framework for other (routing) problems.

**Graph sparsity** We test the two graph sparsification strategies described in Section 3.4 as another way to trade off performance and runtime of DPDP. In Figure 3c, we experiment with different heatmap thresholds from $10^{-5}$ to 0.9 and different values for KNN from 5 to 99 (fully connected).

The heatmap threshold strategy clearly outperforms the KNN strategy as it yields the same results using sparser graphs (and lower runtimes). This illustrates that the heatmap threshold strategy is more informed than the KNN strategy, confirming the value of the neural network predictions.

# 5 Discussion

In this paper we introduced Deep Policy Dynamic Programming, which combines machine learning and dynamic programming for solving vehicle routing problems. The method yields close to optimal results for TSPs with 100 nodes and is competitive to the highly optimized LKH [24] solver for VRPs with 100 nodes. On the TSP with time windows, DPDP also outperforms LKH, being significanlty faster, as well as GVNS [10], the best open source solver we could find. Given that DPDP was not specifically designed for TSPTW, and thus can likely be improved, we consider this an impressive and promising achievement.

The constructive nature of DPDP (combined with search) allows to efficiently address hard constraints such as time windows, which are typically considered challenging in neural combinatorial optimization [5, 31] and are also difficult for local search heuristics (as they need to maintain feasibility while adapting a solution). Given our results on TSP, VRP and TSPTW, and the flexibility of DP as a framework, we think DPDP has great potential for solving many more variants of routing problems, and possibly even other problems that can be formulated using DP (e.g. job shop scheduling [21]). We hope that our work brings machine learning research for combinatorial optimization closer to the operations research (especially vehicle routing) community, by combining machine learning with DP and evaluating the resulting new framework on different data distributions used by different communities [45, 55, 6, 10].

**Scope, limitations & future work** Deep learning for combinatorial optimization is a recent research direction, which could significantly impact the way practical optimization problems get solved in the future. Currently, however, it is still hard to beat most SOTA problem specific solvers from the OR community. Despite our success for TSPTW, DPDP is not yet a practical alternative in general, but we do consider our results as highly encouraging for further research. We believe such research could yield significant further improvement by addressing key current limitations: (1) the scalability to larger instances, (2) the dependency on example solutions and (3) the heuristic nature of the scoring function. First, while 100 nodes is not far from the size of common benchmarks (100 - 1000 for VRP [55] and 20 - 200 for TSPTW [10]), scaling is a challenge, mainly due to the 'fully-connected' $O(n^2)$ graph neural network. Future work could reduce this complexity following e.g. [35]. The dependency on example solutions from an existing solver also becomes more prominent for larger instances, but could potentially be removed by 'bootstrapping' using DP itself as we, in some sense, have done for TSPTW (see Section 3.3). Future work could iterate this process to train the model 'tabula rasa' (without example solutions), where DP could be seen analogous to MCTS in *AlphaZero* [52]. Lastly, the heat + potential score function is a well-motivated but heuristic function that was manually designed as a function of the predicted heatmap. While it worked well for the three problems we considered, it may need suitable adaption for other problems. Training this function end-to-end [11, 62], while keeping a low computational footprint, would be an interesting topic for future work.

# References


[1] Luca Accorsi and Daniele Vigo. A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems. Technical report, Tech. rep., University of Bologna, 2020.

[2] David Applegate, Robert Bixby, Vasek Chvatal, and William Cook. Concorde TSP solver, 2006.

[3] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[4] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

[5] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[6] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. *arXiv preprint arXiv:2006.01610*, 2020.

[7] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 32, pages 6281–6292, 2019.

[8] William Cook and Paul Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.

[9] Paulo Roberto de O da Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. *Proceedings of Machine Learning Research*, 1:17, 2020.

[10] Rodrigo Ferreira Da Silva and Sebastián Urrutia. A general vns heuristic for the traveling salesman problem with time windows. *Discrete Optimization*, 7(4):203–211, 2010.

[11] Hal Daumé III and Daniel Marcu. Learning as search optimization: Approximate large margin methods for structured prediction. In *Proceedings of the 22nd international conference on Machine learning*, pages 169–176, 2005.

[12] Arthur Delarue, Ross Anderson, and Christian Tjandraatmadja. Reinforcement learning with combinatorial actions: An application to vehicle routing. *Advances in Neural Information Processing Systems*, 33, 2020.

[13] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the TSP by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181. Springer, 2018.

[14] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[15] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995.

[16] Jonas K Falkner and Lars Schmidt-Thieme. Learning to solve vehicle routing problems with time windows through joint attention. *arXiv preprint arXiv:2006.09100*, 2020.

[17] Zhang-Hua Fu, Kai-Bin Qiu, and Hongyuan Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. *arXiv preprint arXiv:2012.10658*, 2020.

[18] Lei Gao, Mingxiang Chen, Qichang Chen, Ganzhong Luo, Nuoyi Zhu, and Zhixin Liu. Learn to design the heuristics for vehicle routing problem. *arXiv preprint arXiv:2002.08539*, 2020.

[19] Maxime Gasse, Didier Chetelat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*.

[20] Joaquim Gromicho, Jelke J van Hoorn, Adrianus Leendert Kok, and Johannes MJ Schutten. Restricted dynamic programming: a flexible framework for solving realistic vrps. *Computers & operations research*, 39(5):902–909, 2012.

[21] Joaquim AS Gromicho, Jelke J Van Hoorn, Francisco Saldanha-da Gama, and Gerrit T Timmer. Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research*, 39(12):2968–2977, 2012.

[22] Gurobi Optimization, LLC. Gurobi, 2018.

[23] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[24] Keld Helsgaun. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems: Technical report. 2017.

[25] André Hottung, Bhanu Bhandari, and Kevin Tierney. Learning a latent search space for routing problems using variational autoencoders. In *International Conference on Learning Representations*, 2021.

[26] André Hottung and Kevin Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. *arXiv preprint arXiv:1911.09539*, 2019.

[27] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019.

[28] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. On learning paradigms for the travelling salesman problem. *arXiv preprint arXiv:1910.07210*, 2019.

[29] Minsu Kim, Jinkyoo Park, and Joungho Kim. Learning collaborative policies to solve np-hard routing problems. In *Advances in Neural Information Processing Systems*, 2021.

[30] AL Kok, Elias W Hans, Johannes MJ Schutten, and Willem HM Zijm. A dynamic programming heuristic for vehicle routing with time-dependent travel times and required breaks. *Flexible services and manufacturing journal*, 22(1-2):83–108, 2010.

[31] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

[32] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33, 2020.

[33] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European journal of operational research*, 59(3):345–358, 1992.

[34] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[35] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[36] Sirui Li, Zhongxia Yan, and Cathy Wu. Learning to delegate for large-scale vehicle routing. In *Advances in Neural Information Processing Systems*, 2021.

[37] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in Neural Information Processing Systems*, page 539, 2018.

[38] Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2020.

[39] Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *arXiv preprint arXiv:1911.04936*, 2019.

[40] Yining Ma, Jingwen Li, Zhiguang Cao, Wen Song, Le Zhang, Zhenghua Chen, and Jing Tang. Learning to iteratively solve routing problems with dual-aspect collaborative transformer. In *Advances in Neural Information Processing Systems*, 2021.

[41] Chryssi Malandraki and Robert B Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *European Journal of Operational Research*, 90(1):45–55, 1996.

[42] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *arXiv preprint arXiv:2003.03600*, 2020.

[43] Aristide Mingozzi, Lucio Bianco, and Salvatore Ricciardelli. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. *Operations research*, 45(3):365–377, 1997.

[44] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[45] MohammadReza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takac. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9860–9870, 2018.

[46] Clara Novoa and Robert Storer. An approximate dynamic programming approach for the vehicle routing problem with stochastic demands. *European Journal of Operational Research*, 196(2):509–515, 2009.

[47] Alex Nowak, Soledad Villar, Afonso S Bandeira, and Joan Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint arXiv:1706.07450*, 2017.

[48] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[49] Bo Peng, Jiahai Wang, and Zizhen Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. In *International Symposium on Intelligence Computation and Applications*, pages 636–650. Springer, 2019.

[50] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

[51] Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

[52] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[53] Yuan Sun, Andreas Ernst, Xiaodong Li, and Jake Weiner. Generalization of machine learning for problem reduction: a case study on travelling salesman problems. *OR Spectrum*, pages 1–27, 2020.

[54] Paolo Toth and Daniele Vigo. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

[55] Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017.

[56] Wouter van Heeswijk and Han La Poutré. Approximate dynamic programming with neural networks in linear discrete action spaces. *arXiv preprint arXiv:1902.09855*, 2019.

[57] Jelke J. van Hoorn. *Dynamic Programming for Routing and Scheduling*. PhD thesis, 2016.

[58] Natalia Vesselinova, Rebecca Steinert, Daniel F Perez-Ramirez, and Magnus Boman. Learning combinatorial optimization on graphs: A survey with applications to networking. *IEEE Access*, 8:120388–120416, 2020.

[59] Thibaut Vidal. Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. *arXiv preprint arXiv:2012.10384*, 2020.

[60] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, Nadia Lahrichi, and Walter Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research*, 60(3):611–624, 2012.

[61] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[62] Sam Wiseman and Alexander M Rush. Sequence-to-sequence learning as beam-search optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1296–1306, 2016.

[63] Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems. *arXiv preprint arXiv:1912.05784*, 2019.

[64] Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. Step-wise deep learning models for solving routing problems. *IEEE Transactions on Industrial Informatics*, 2020.

[65] Liang Xin, Wen Song, Zhiguang Cao, and Jie Zhang. NeuroLKH: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. In *Advances in Neural Information Processing Systems*, 2021.

[66] Shenghe Xu, Shivendra S Panwar, Murali Kodialam, and TV Lakshman. Deep neural network approximated dynamic programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1684–1691, 2020.

[67] Feidiao Yang, Tiancheng Jin, Tie-Yan Liu, Xiaoming Sun, and Jialin Zhang. Boosting dynamic programming with neural networks for solving np-hard problems. In *Asian Conference on Machine Learning*, pages 726–739. PMLR, 2018.

# A Implementation

We implement the dynamic programming algorithm on the GPU using PyTorch [48]. While mostly used as a Deep Learning framework, it can be used to speed up generic (vectorized) computations.

## A.1 Beam variables

For each solution in the beam, we keep track of the following variables (storing them for all solutions in the beam as a vector): the cost, current node, visited nodes and (for VRP) the remaining capacity or (for TSPTW) the current time. As explained, these variables can be computed incrementally when generating expansions. Additionally, we keep a variable vector *parent*, which, for each solution in the current beam, tracks the index of the solution in the previous beam that generated the expanded solution. To compute the score of the policy for expansions efficiently, we also keep track of the score for each solution and the potential for each node for each solution incrementally.

We do not keep past beams in memory, but at the end of each iteration, we store the vectors containing the parents as well as last actions for each solution on the *trace*. As the solution is completely defined by the sequence of actions, this allows to backtrack the solution after the algorithm has finished. To save GPU memory (especially for larger beam sizes), we store the $O(Bn)$ sized trace on the CPU memory.

For efficiency, we keep the set of visited nodes as a bitmask, packed into 64-bit long integers (2 for 100 nodes). Using bitwise operations with the packed adjacency matrix, this allows to quickly check feasible expansions (but we need to *unpack* the mask into boolean vectors to find all feasible expansions explicitly). Figure 4a shows an example of the beam (with variables related to the policy and backtracking omitted) for the VRP.

## A.2 Generating non-dominated expansions

A solution $\boldsymbol{a}$ can only dominate a solution $\boldsymbol{a}'$ if $\text{visited}(\boldsymbol{a}) = \text{visited}(\boldsymbol{a}')$ and $\text{current}(\boldsymbol{a}) = \text{current}(\boldsymbol{a}')$, i.e. if they correspond to the same *DP state*. If this is the case, then, if we denote by $\text{parent}(\boldsymbol{a})$ the parent solution from which $\boldsymbol{a}$ was expanded, it holds that

$$\begin{aligned}\text{visited}(\text{parent}(\boldsymbol{a})) &= \text{visited}(\boldsymbol{a}) \setminus \{\text{current}(\boldsymbol{a})\} \\ &= \text{visited}(\boldsymbol{a}') \setminus \{\text{current}(\boldsymbol{a}')\} \\ &= \text{visited}(\text{parent}(\boldsymbol{a}')).\end{aligned}$$

This means that only expansions from solutions with the same set of visited nodes can dominate each other, so we only need to check for dominated solutions among groups of expansions originating from parent solutions with the same set of visited nodes. Therefore, before generating the expansions, we group the current beam (the parents of the expansions) by the set of visited nodes (see Figure 4a). This can be done efficiently, e.g. using a lexicographic sort of the packed bitmask representing the sets of visited nodes[10].

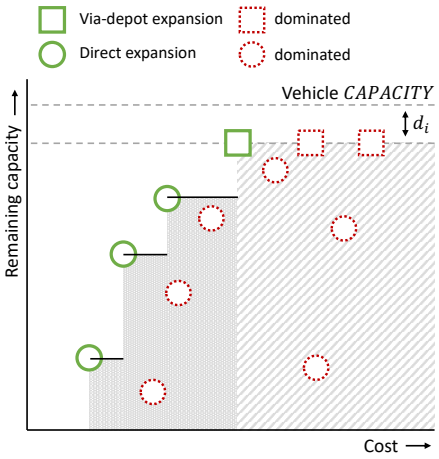
### A.2.1 Travelling Salesman Problem

For TSP, we can generate (using boolean operations) the $B \times n$ matrix with boolean entries indicating feasible expansions (with $n$ action columns corresponding to $n$ nodes, similar to the $B \times 2n$ matrix for

[10] For efficiency, we use a custom function similar to TORCH.UNIQUE, and argsort the returned inverse after which the resulting permutation is applied to all variables in the beam.

| Cost | Capacity | Visited | Current | Direct 0 | 1 | 2 | 3 | 4 | Via-depot 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 5 | 01101 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 12 | 8 | 01101 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 13 | 7 | 01101 | 2 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 3 | 01101 | 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 11 | 7 | 10101 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 12 | 6 | 10101 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 13 | 7 | 10101 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

(a) Example beam for VRP with variables, grouped by set of visited nodes (left) and feasible, non-dominated expansions (right), with $2n$ columns corresponding to $n$ direct expansions and $n$ via-depot expansions. Some expansions to unvisited nodes are infeasible, e.g. due to the capacity constraint or a sparse adjacency graph. The shaded areas indicate groups of candidate expansions among which dominances should be checked: for each set of visited nodes there is only one non-dominated via-depot expansion (indicated by solid green square), which must necessarily be an expansion of the solution that has the lowest cost to return to the depot (indicated by the dashed green rectangle ; note that the cost displayed excludes the cost to return to the depot). Direct expansions can be dominated (indicated by red dotted circles) by the single non-dominated via-depot expansion or other direct expansions with the same DP state (set of visited nodes and expanded node, as indicated by the shaded areas). See also Figure 4b for (non-)dominated expansions corresponding to the same DP state.




(b) Example of a set of dominated and non-dominated expansions (direct and via-depot) corresponding to the same DP state (set of visited nodes and expanded node $i$) for VRP. Non-dominated expansions have lower cost or higher remaining capacity compared to all other expansions. The right striped area indicates expansions dominated by the (single) non-dominated via-depot expansion. The left (darker) areas are dominated by individual direct expansions. Dominated expansions in this area have remaining capacity lower than the cumulative maximum remaining capacity when going from left to right (i.e. in sorted order of increasing cost), indicated by the black horizontal lines.

Figure 4: Implementation of DPDP for VRP

VRP in Figure 4a), i.e. nodes that are unvisited and adjacent to the current node. If we find positive entries sequentially for each column (e.g. by calling TORCH.NONZERO on the transposed matrix), we get all expansions grouped by the combination of action (new current node) and parent set of visited nodes, i.e. grouped by the DP state. We can then trivially find the segments of consecutive expansions corresponding to the same DP state, and we can efficiently find the minimum cost solution for each segment, e.g. using TORCH_SCATTER [11].

### A.2.2 Vehicle Routing Problem

For VRP, the dominance check has two dimensions (cost *and* remaining capacity) and additionally we need to consider $2n$ actions: $n$ direct and $n$ via the depot (see Figure 4a). Therefore, as we will explain, we check dominances in two stages: first we find (for each DP state) the *single* non-dominated 'via-depot' expansion, after which we find all non-dominated 'direct' expansions (see Figure 4b).

The DP state of each expansion is defined by the expanded node (the new current node) and the set of visited nodes. For each DP state, there can be only *one*[12] non-dominated expansion where the last action was via the depot, since all expansions resulting from 'via-depot actions' have the same remaining capacity as visiting the depot resets the capacity (see Figure 4b). To find this expansion, we first find, for each unique set of visited nodes in the current beam, the solution that can return to the depot with lowest total cost (thus including the cost to return to the depot, indicated by a dashed green rectangle in Figure 4a). The single non-dominated 'via-depot expansion' for each DP state must necessarily be an expansion of this solution. Also observe that this via-depot solution cannot be dominated by a solution expanded using a direct action, which will always have a lower remaining vehicle capacity (assuming positive demands) as can bee seen in Figure 4b. We can thus generate the non-dominated via-depot expansion for each DP state efficiently and independently from the direct expansions.

For each DP state, all *direct* expansions with cost higher (or equal) than the via-depot expansion can directly be removed since they are dominated by the via-depot expansion (having higher cost and lower remaining capacity, see Figure 4b). After that, we sort the remaining (if any) direct expansions for each DP state based on the cost (using a segmented sort as the expansions are already grouped if we generate them similarly to TSP, i.e. per column in Figure 4a). For each DP state, the lowest cost solution is never dominated. The other solutions should be kept only if their remaining capacity is strictly larger than the largest remaining capacity of all lower-cost solutions corresponding to the same DP state, which can be computed using a (segmented) cumulative maximum computation (see Figure 4b).

### A.2.3 TSP with Time Windows

For the TSPTW, the dominance check has two dimensions: cost and time. Therefore, it is similar to the check for non-dominated direct expansions for the VRP (see Figure 4b), but replacing remaining capacity (which should be maximized) by current time (to be minimized). In fact, we could reuse the implementation, if we replace remaining capacity by time multiplied by $-1$ (as this should be minimized). This means that we sort all expansions for each DP state based on the cost, keep the first solution and keep other solutions only if the time is strictly lower than the lowest current time for all lower-cost solutions, which can be computed using a cumulative minimum computation.

[11] `https://github.com/rusty1s/pytorch_scatter`

[12] Unless we have multiple expansions with the same costs, in which case can pick one arbitrarily.

## A.3 Finding the top $B$ solutions

We may generate all 'candidate' non-dominated expansions and then select the top $B$ using the score function. Alternatively, we can generate expansions in batches, and keep a streaming top $B$ using a priority queue. We use the latter implementation, where we can also derive a bound for the score as soon as we have $B$ candidate expansions. Using this bound, we can already remove solutions before checking dominances, to achieve some speedup in the algorithm.[13]

## A.4 Performance improvements

There are many possibilities for improving the speed of the algorithm. For example, PyTorch lacks a segmented sort so we use a much slower lexicographic sort instead. Also an efficient GPU priority queue would allow much speedup, as we currently use sorting as PyTorch' top-$k$ function is rather slow for large $k$. In some cases, a binary search for the $k$-th largest value can be faster, but this introduces undesired CUDA synchronisation points. We currently use multiprocessing to solve multiple instances on a single GPU in parallel, introducing a lot of Python overhead. A batched implementation would give a significant speedup.

[13] This may give slightly different results if the scoring function is inconsistent with the domination rules, i.e. if a better scoring solution would be dominated by a worse scoring solution but is not since that solution is removed using the score bound before checking the dominances.

# B TSP with Time Windows

This section contains additional information for the TSPTW.

## B.1 Adaption of model for TSPTW

The model updates the edge embedding $e_{ij}^l$ for edge $(i, j)$ in layer $l + 1$ using node embeddings $x_i^l$ and $x_j^l$ with the following equation (Equation (5) in [27]):

$$e_{ij}^{l+1} + \text{ReLU}(\text{BatchNorm}(W_3^l e_{ij}^l + W_4^l x_i^l + W_5^l x_j^l)) \tag{2}$$

where $W_3^l, W_4^l$ and $W_5^l$ are trainable parameters. We found out that the implementation[14] actually shares the parameters $W_4^l$ and $W_5^l$, i.e. $W_4^l = W_5^l$, resulting in $e_{ij}^l = e_{ji}^l$ for all layers $l$, as for $l = 0$ both directions are initialized the same. To allow the model to make different predictions for different directions, we implement $W_5^l$ as a separate parameter, such that the model can have different representations for edges $(i, j)$ and $(j, i)$. We define the training labels accordingly for directed edges, so if edge $(i, j)$ is in the directed solution, it will have a label 1 whereas the edge $(j, i)$ will not (for the undirected TSP and VRP, both labels are 1).

## B.2 Dataset generation

We found that using our DP formulation for TSPTW, the instances by [6] were all solved optimally, even with a very small beam size (around 10). This is because there is very little overlap in the time windows as a result of the way they are generated, and therefore very few actions are feasible as most of the actions would 'skip over other time windows' (advance the time so much that other nodes can no longer be served)[15]. We conducted some quick experiments with a weaker DP formulation, that only checks if actions *directly* violate time windows, but does not check if an action causes other nodes to be no longer reachable in their time windows. Using this formulation, the DP algorithm can run into many dead ends if just a single node gets skipped, and using the GNN policy (compared to a cost based policy as in Section 4.4) made the difference between good solutions and no solution at all being found.

We made two changes to the data generation procedure by [6] to increase the difficulty and make it similar to [10], defining the 'large time window' dataset. First, we sample the time windows around arrival times when visiting nodes in a random order without any waiting time, which is different from [6] who 'propagate' the waiting time (as a result of time windows sampled). Our modification causes a tighter schedule with more overlap in time windows, and is similar to [10]. Secondly, we increase the maximum time window size from 100 to 1000, which makes that the time windows are in the order of 10% of the horizon[16]. This doubles the maximum time window size of 500 used by [10] for instances with 200 nodes, to compensate for half the number of nodes that can possibly overlap the time window.

To generate the training data, for practical reasons we used DP with the heuristic 'cost heat + potential' strategy and a large beam size (1M), which in many cases results in optimal solutions being found.

[14] `https://github.com/chaitjo/graph-convnet-tsp/blob/master/models/gcn_layers.py`

[15] If all time windows are disjoint, there is only one feasible solution. Therefore, the amount of overlap in time windows determines to some extent the 'branching factor' of the problem and the difficulty.

[16] Serving 100 customers in a 100x100 grid, empirically we find the total schedule duration including waiting (the makespan) is around 5000.