Deep Policy Dynamic Programming for Vehicle Routing Problems

Wouter Kool[⋆1,2][0000−0002−1837−1454],
Herke van Hoof[1][0000−0002−1583−3692],
Joaquim Gromicho[1,2][0000−0002−1467−6656], and
Max Welling[1][0000−0003−1484−2121]

[1] University of Amsterdam, The Netherlands
[2] ORTEC, The Netherlands

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.

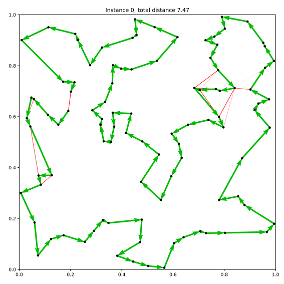
Keywords: Dynamic Programming · Deep Learning · Vehicle Routing.

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

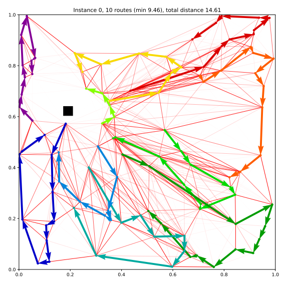
1 Introduction

Dynamic programming (DP) [7] is a powerful framework for solving optimization problems by solving smaller subproblems through the principle of optimality [4]. Famous examples are Dijkstra's algorithm [16] for the shortest route between two locations, and the classic Held-Karp algorithm for the travelling salesman problem (TSP) [26, 5]. 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 [64, 6, 50, 36, 9]. While the results are promising, most learned heuristics are not (yet) competitive to 'traditional' algorithms such as LKH [27] and lack (asymptotic) guarantees on their performance.

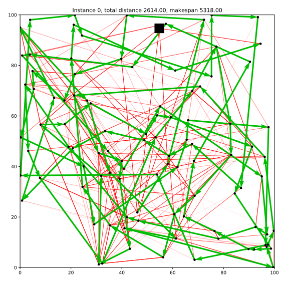
In this paper, we propose *Deep Policy Dynamic Programming* (DPDP) as a framework for solving vehicle routing problems. The key of DPDP is to combine the strengths of deep learning and DP, by restricting the DP state space (the search space) using a policy derived from a neural network. In Figure 1 it can be seen how the neural network indicates promising parts of the search space as a *heatmap* over the edges of the graph. This heatmap used by the DP algorithm to find a good solution. DPDP is more powerful than some related ideas [70, 25, 69, 8, 42] as it combines 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 [22]. We illustrate this by testing DPDP on the TSP, the capacitated VRP and the TSP with (hard) time window constraints (TSPTW).



(a) Travelling Salesman Problem



(b) Vehicle Routing Problem



(c) TSP with Time Windows

Fig. 1: Heatmap predictions (red) and solutions (colored) by DPDP (VRP depot edges omitted for clarity). The heatmap indicates only a small fraction of all edges as promising, while including (almost) all edges from the solution.

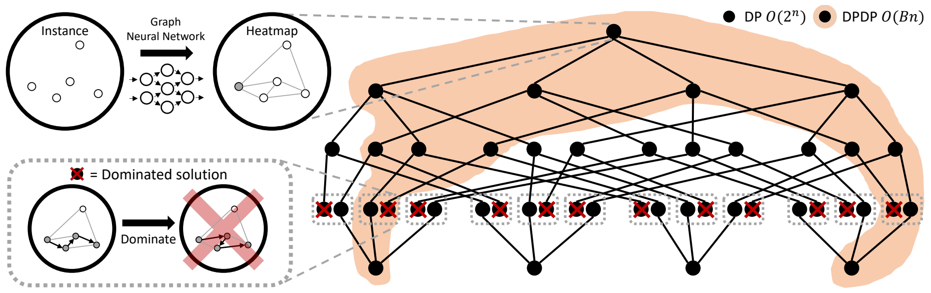


Fig. 2: DPDP for the TSP. A GNN creates a (sparse) heatmap indicating promising edges, after which a tour is constructed using forward dynamic programming. In each step, at most B solutions are expanded according to the heatmap policy, restricting the size of the search space. Partial solutions are dominated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

In more detail, the starting point of our proposed approach is a *restricted dynamic programming* algorithm [46, 22], which heuristically reduces the search space by retaining at most B solutions per iteration. The selection process is important as it defines the part of the DP state space considered and, thus, the quality of the solution found (see Fig. 2). DPDP defines the selection using a (sparse) heatmap of promising route segments, obtained by pre-processing the problem instance using a (deep) graph neural network (GNN) [32]. This brings the power of neural networks to DP, inspired by the success of neural networks that improved tree search [57] or branch-and-bound algorithms [21, 49].

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 [64, 6, 15, 36], 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 [32], 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 [27] 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 [12], illustrating the power of DPDP to handle difficult hard constraints (time windows).

2 Related work

DP [7] has a long history as an exact solution method for routing problems [38, 59], e.g. the TSP with time windows [17] and precedence constraints [48], 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 [46], and has been generalized into a flexible framework for VRPs with different types of practical constraints [22]. DP approaches have also been shown to be useful in settings with difficult practical issues such as time-dependent travel times and driving regulations [35] or stochastic demands [51]. For more examples of DP for routing (and scheduling), see [28]. For sparse graphs, alternative, but less flexible, formulations can be used [10].

Despite the flexibility, DP methods have not gained much popularity compared to heuristic approaches such as R&R [56], ALNS [55], LKH [27], HGS [63, 62] 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 [22], 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 consider, which is the motivation behind DPDP.

In the machine learning community, deep neural networks (DNNs) have recently boosted performance on various tasks [39]. After the first DNN model was trained (using example solutions) to construct TSP tours [64], many improvements have been proposed, e.g. different training strategies such as reinforcement learning (RL) [6, 33, 14, 37] and model architectures, which enabled the same idea to be used for other routing problems [50, 36, 15, 54, 18, 67, 45]. 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* [52, 32, 19], 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 [9, 43, 20, 66, 30, 34, 41, 68, 29]. Scaling to instances beyond 100 nodes remains challenging [44, 19].

The combination of machine learning with DP has been proposed in limited settings [70, 25, 69]. Most related to our approach, a DP algorithm for TSPTW, guided by an RL agent, was implemented using an existing solver [8], 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 and local search for maximum independent set (MIS) [42]. Whereas DPDP directly builds on the idea of predicting promising edges [42, 32], it uses these more efficiently through a policy with *potential function* (see Section 3.2), 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 [47, 61, 3].

3 Deep Policy Dynamic Programming

DPDP uses an existing graph neural network [32], suitably adapted for VRP and TSPTW, to predict a heatmap of promising edges. This heatmap is used in the DP algorithm in two ways: 1) to exclude edges with a value below the *heatmap threshold* of 10^{-5} from the graph and 2) to define a *scoring policy* to select candidate solutions in each iteration. In more detail, as illustrated in Fig. 2, the DP algorithm starts with a *beam* of a single initial (empty) solution, and 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. 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*, with *beam size* B. This is different from a 'standard' beam search, which considers the *solution space* by not removing dominated solutions. DPDP is asymptotically optimal as using $B = n \cdot 2^n$ for a TSP with n nodes guarantees optimal results, but by choosing a smaller B, DPDP can trade off 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) the **scoring policy**, based on the neural network, for selecting the B solutions to keep. A solution is always defined by a sequence of actions, which allows the DP algorithm to construct the final solution by backtracking. In the next sections, we describe the neural network and define the DPDP ingredients for the TSP, VRP and TSPTW.

3.1 The graph neural network

We use the original (pre-trained) model from [32] (which we describe in detail in Appendix 1 for self-containment) for the TSP, but we modify the neural network architecture and train new models to support the VRP and TSPTW, as we describe in Sections 3.3 and 3.4. In general, the resulting model uses problem-specific node input features and edge input features, which get transformed into initial representations of the nodes and edges. These representations then get updated sequentially using a number of *graph convolutional layers*, which exchange information between the nodes and edges. The final edge representation is used to make the prediction whether the edge is promising, i.e. whether it has a high probability of being part of the optimal solution.

The model is trained using a large training dataset of problem instances with optimal (or high-quality) solutions, obtained using an existing solver. While it takes a significant amount of resources to create this dataset and train the model (each of which can take up to a number of days on a single machine), training of the model is, in principle, only required once given a specific distribution of problem instances. We consider only instances with $n = 100$ nodes, but the model can handle instances of different graph sizes, although good generalization may be limited to graphs with sizes close to the size trained for [36, 33].

3.2 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 nodes 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[3]. A solution can only dominate other solutions with the same set of visited nodes, so we only need to remove dominated solutions from sets of solutions with the same number of actions. This is why the DP algorithm can be executed in iterations (as explained): at step t all solutions in the beam have t actions and $t + 1$ visited nodes (including the start node). The resulting memory need is thus limited to $O(B)$ states, with B the beam size.

We define the **scoring policy** using the pretrained model from [32], 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

[3] 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.

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 neighbors 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. It is a heuristic estimate of the total heat of the complete solution, but it is not an estimate of the cost objective (which has a different unit), neither it is a *bound* on the total heat or cost objective.

3.3 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 VRP solution has desirable properties 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 $\text{visited}(\boldsymbol{a}^*) = \text{visited}(\boldsymbol{a})$ and $\text{current}(\boldsymbol{a}^*) = \text{current}(\boldsymbol{a})$ (i.e. $\boldsymbol{a}^*$ corresponds to the same DP state) and $\text{cost}(\boldsymbol{a}^*) \leq \text{cost}(\boldsymbol{a})$ and $\text{capacity}(\boldsymbol{a}^*) \geq \text{capacity}(\boldsymbol{a})$, 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 [32] (described in Appendix 1) to include the depot node and demands. We mark the depot as a special node type, which affects the initial node representation similarly to edge types, and we add additional edge types for connections to the depot. Additionally, each node gets an extra input (next to its coordinates) corresponding to $d_i/\text{CAPACITY}$ (where we set $d_{\text{DEP}} = 0$). The model is trained on example solutions from LKH [27] (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 heat of the initial state 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.

3.4 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}, i.e. we assume a speed of 1 (w.l.o.g. as we can rescale time). 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[4]. 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 ($\text{visited}(\boldsymbol{a}^*) = \text{visited}(\boldsymbol{a})$ and $\text{current}(\boldsymbol{a}^*) = \text{current}(\boldsymbol{a})$) and is strictly better in terms of cost and time, i.e. $\text{cost}(\boldsymbol{a}^*) \leq \text{cost}(\boldsymbol{a})$ and $\text{time}(\boldsymbol{a}^*) \leq \text{time}(\boldsymbol{a})$, with *at least one of the two inequalities being strict.*

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

The model [32] for the **scoring policy** is adapted to include the time windows (l_i, u_i) as node features (scaled to correspond to a speed of 1 for the input distances and coordinates, which are scaled to the range $[0, 1]$), 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 1). 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.

4 Experiments

We implement DPDP using PyTorch [53] to leverage GPU computation. For details, see Appendix 2. Our code is publicly available.[5] 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.

4.1 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 [36]. We report cost and *gap* to the optimal solution found using Concorde [2] (following [36]) and compare against LKH [27] and Gurobi [24], 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 [19]. Times for generating heatmaps (if applicable) is reported separately (as the first term) from the running time for MCTS [19] or DP. DPDP achieves close to optimal results, strictly outperforming the neural baselines achieving better results in less time (except the Attention Model trained with POMO [37], 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 [50] and solved using one run of LKH [27]. 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 [32]) from which we select the saved checkpoint with the lowest validation loss. We use the validation and test sets by [36].

[5] https://github.com/wouterkool/dpdp

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 [63, 62]				15.563	0.000 %	6H11M
GUROBI [24]	7.776	0.151 %	31M			
LKH [27]	7.765	0.000 %	42M	15.647	0.536 %	12H57M
GNN HEATMAP + BEAM SEARCH [32]	7.87	1.39 %	40M			
LEARNING 2-OPT HEURISTICS [11]	7.83	0.87 %	41M			
MERGED GNN HEATMAP + MCTS [19]	7.764*	0.04 %	4M + 11M			
ATTENTION MODEL + SAMPLING [36]	7.94	2.26 %	1H	16.23	4.28 %	2H
STEP-WISE ATTENTION MODEL [67]	8.01	3.20 %	29S	16.49	5.96 %	39S
ATTN. MODEL + COLL. POLICIES [34]	7.81	0.54 %	12H	15.98	2.68 %	5H
LEARNING IMPROV. HEURISTICS [66]	7.87	1.42 %	2H	16.03	3.00 %	5H
DUAL-ASPECT COLL. TRANSFORMER [45]	7.77	0.09 %	5H	15.71	0.94 %	9H
ATTENTION MODEL + POMO [37]	7.77	0.14 %	1M	15.76	1.26 %	2M
NEUREWRITER [9]				16.10	3.45 %	1H
DYNAMIC ATTN. MODEL + 2-OPT [54]				16.27	4.54 %	6H
NEUR. LRG. NEIGHB. SEARCH [30]				15.99	2.74 %	1H
LEARN TO IMPROVE [43]				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

Table 1 shows the results, where the gap is relative to Hybrid Genetic Search (HGS)[6], a SOTA heuristic VRP solver [63, 62]. HGS is faster and improves around 0.5% over LKH [27], which is typically considered the baseline in related work. We present the results for LKH, as well as the strongest neural approaches and DPDP with beam sizes up to 1 million. Some results used 2000 (different) instances [43] and cannot be directly compared[7]. DPDP outperforms all other neural baselines, except the Attention Model trained with POMO [37], 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 [60]. 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[8]. 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.

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 [63, 62]	18050	0.000 %	7h53m	3h56m
LKH [27]	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

[6] https://github.com/vidalt/HGS-CVRP

[7] The running time of 4000 hours (167 days) is estimated from 24min/instance [43].

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

4.3 TSP with Time Windows

For the TSP with hard time window constraints, we use the data distribution by [8] 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 [12], which has larger time windows which are more difficult as the feasible search space is larger[9] [17]. For details, see Appendix 1. 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 Neighborhood Search (GVNS) [12], the best open-source solver for TSPTW we could find[10], 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* [8], 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.

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

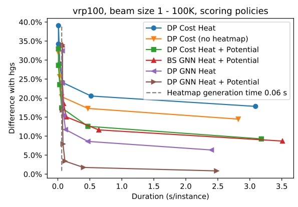
Problem	Small time windows [8] (100 inst.)				Large time windows [12] (10k inst.)			
Method	Cost	Gap	Fail	Time	Cost	Gap	Fail	Time
GVNS 30x [12]	5129.58	0.000 %		7s	2432.112	0.000 %		37m15s
GVNS 1x [12]	5129.58	0.000 %		<1s	2457.974	1.063 %		1m4s
LKH 1x [27]	5130.32	0.014 %	1.00 %	5m48s	2431.404	-0.029 %		34h58m
BaB-DQN* [8]	5130.51	0.018 %		25h				
ILDS-DQN* [8]	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

[9] Up to a limit, as making the time windows infinite size reduces the problem to plain TSP.

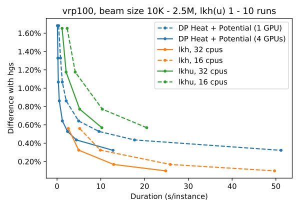
[10] https://github.com/sashakh/TSPTW

4.4 Ablations

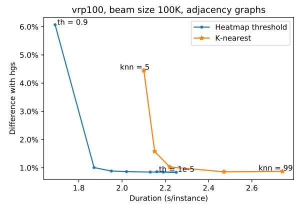
Scoring policy To evaluate the value of different components of DPDP's **GNN Heat + Potential** scoring policy, we compare against other variants. **GNN Heat** is the version without the potential, whereas **Cost Heat + Potential** and **Cost Heat** are variants that use a 'heuristic' $\hat{h}_{ij} = \frac{c_{ij}}{\max_k c_{ik}}$ 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 (no heatmap threshold). 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.



(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.

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

Beam size With DPDP, we can trade off the performance vs. the runtime using the beam size B (and the graph sparsity, see below). Figure 3b illustrates this trade-off, 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 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, analogous to how results are reported in Table 1. We emphasize that in most related work (e.g. [36]), 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 Using the heatmap threshold, the DP algorithm uses a sparse graph to define feasible expansions, which reduces the runtime but may also sacrifice solution quality. For most edges, the model confidently predicts close to 0, such that they are ruled out, even using the default (low) heatmap threshold of 10^{-5}. We may rule out even more edges by increasing the threshold, which can be seen as a secondary way (besides varying the beam size) to trade off the performance and computational cost of DPDP. While this can be seen as a form of learned *problem reduction* [58], we also consider a heuristic alternative of using the K-nearest neighbor (KNN) graph.[11] 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 [27] solver for VRPs with 100 nodes. On the TSPTW, DPDP also outperforms LKH, being significantly faster, as well as GVNS [12], 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) naturally supports hard constraints such as time windows, which are typically considered challenging in neural combinatorial optimization [6, 36] 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 [23]). 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 [50, 60, 8, 12].

[11] For the symmetric TSP and VRP, we add KNN edges in both directions. For the VRP, we also connect each node to the depot (and vice versa) to ensure feasibility.

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 [60] and 20 - 200 for TSPTW [12]), 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. [40]. 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 3.4). 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* [57]. 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 [13, 65], while keeping a low computational footprint, would be an interesting topic for future work.

Acknowledgements

We would like to thank Jelke van Hoorn and Johan van Rooij for helpful discussions. Also we would like to thank anonymous reviewers for helpful suggestions. This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

Appendix 1 The graph neural network model

For the TSP, we use the exact model from [32], which we describe here for self-containment. The model uses node input features and edge input features, which get transformed into initial representations of the nodes and edges. These representations then get updated sequentially using a number of graph convolutional layers, which exchange information between nodes and edges, after which the final edge representation is used to predict whether the edge is part of the optimal solution.

Input features and initial representation The model uses input features for the nodes, consisting of the (x, y)-coordinates, which are then projected into H-dimensional initial embeddings $\mathbf{x}_i^0$ ($H = 300$). The initial edge features $\mathbf{e}_{ij}^0$ are a concatenation of a $\frac{H}{2}$-dimensional projection of the cost (Euclidean distance) c_{ij} from i to j, and a $\frac{H}{2}$-dimensional embedding of the edge type: 0 for normal edges, 1 for edges connecting K-nearest neighbors ($K = 20$) and 2 for *self-loop* edges connecting a node to itself (which are added for ease of implementation).

Graph convolutional layers In each of the $L = 30$ layers of the model, the node and edge representations $\mathbf{x}_i^\ell$ and $\mathbf{e}_{ij}^\ell$ get updated into $\mathbf{x}_i^{\ell+1}$ and $\mathbf{e}_{ij}^{\ell+1}$ [32]:

$$\mathbf{x}_i^{\ell+1} = \mathbf{x}_i^\ell + \mathrm{ReLU}\left(\mathrm{BN}\left(W_1^\ell \mathbf{x}_i^\ell + \sum_{j \in \mathcal{N}(i)} \frac{\sigma(\mathbf{e}_{ij}^\ell)}{\sum_{j' \in \mathcal{N}(i)} \sigma(\mathbf{e}_{ij'}^\ell)} \odot W_2^\ell \mathbf{x}_j^\ell\right)\right) \tag{2}$$

$$\mathbf{e}_{ij}^{\ell+1} = \mathbf{e}_{ij}^\ell + \mathrm{ReLU}\left(\mathrm{BN}\left(W_3^\ell \mathbf{e}_{ij}^\ell + W_4^\ell \mathbf{x}_i^\ell + W_5^\ell \mathbf{x}_j^\ell\right)\right). \tag{3}$$

Here $\mathcal{N}(i)$ is the set of neighbors of node i (in our case all nodes, including i, as we use a fully connected input graph), $\odot$ is the element-wise product and σ is the sigmoid function, applied element-wise to the vector $\mathbf{e}_{ij}^\ell$. $\mathrm{ReLU}(\cdot) = \max(\cdot, 0)$ is the rectified linear unit and BN represents batch normalization [31]. W_1, W_2, W_3, W_4 and W_5 are trainable parameter matrices, where we fix $W_4 = W_5$ for the symmetric TSP.

Output prediction After L layers, the final prediction $h_{ij} \in (0, 1)$ is made independently for each edge (i, j) using a multi-layer perceptron (MLP), which takes $\mathbf{e}_{ij}^L$ as input and has two H-dimensional hidden layers with ReLU activation and a 1-dimensional output layer, with sigmoid activation. We interpret h_{ij} as the predicted probability that the edge (i, j) is part of the optimal solution, which indicates how promising this edge is when searching for the optimal solution.

Training For TSP, the model is trained on a dataset of 1 million optimal solutions, found using Concorde [2], for randomly generated TSP instances. The training loss is a weighted binary cross-entropy loss, that maximizes the prediction quality when h_{ij} is compared to the ground-truth optimal solution. Generating the dataset takes between half a day and a few days (depending on number of CPU cores), and training the model takes a few days on one or multiple GPUs, but both are only required once given a desired data distribution.

1.1 Predicting directed edges for the TSPTW

The TSP is an undirected problem, so the neural network implementation[12] by [32] shares the parameters W_4^l and W_5^l in Equation (3), i.e. $W_4^l = W_5^l$, resulting in $\mathbf{e}_{ij}^l = \mathbf{e}_{ji}^l$ for all layers l, as for $l = 0$ both directions are initialized the same. While the VRP also is an undirected problem, the TSPTW is directed as the direction of the route determines the times of arrival at different nodes. 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).

1.2 Dataset generation for the TSPTW

We found that using our DP formulation for TSPTW, the instances by [8] 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)[13]. 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 [8] to increase the difficulty and make it similar to [12], 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 [8] 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 [12]. 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[14]. This doubles the maximum time window size of 500 used by [12] 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.

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

[13] 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.

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

Appendix 2 Implementation

We implement the dynamic programming algorithm on the GPU using PyTorch [53]. While mostly used as a Deep Learning framework, it can be used to speed up generic (vectorized) computations.

2.1 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.

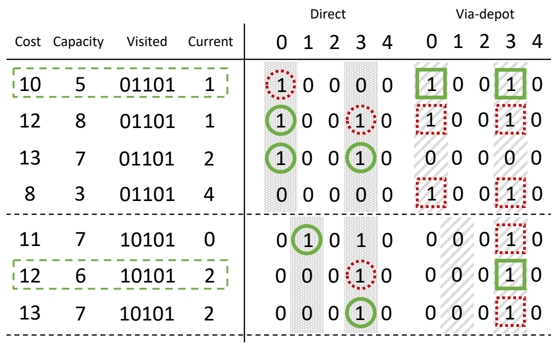
For efficiency, we keep the set of visited nodes as a bitmask, packed into 64-bit long integers (2 for 100 nodes). Using bitwise operations with the packed adjacency matrix, this allows to quickly check feasible expansions (but we need to *unpack* the mask into boolean vectors to find all feasible expansions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

2.2 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

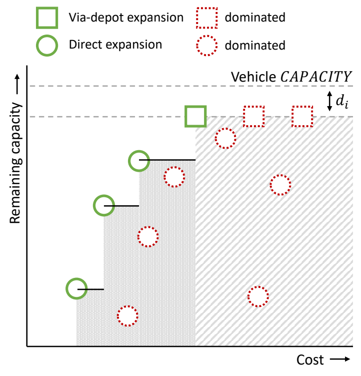
$$\begin{aligned}\text{visited}(\text{parent}(\boldsymbol{a})) &= \text{visited}(\boldsymbol{a}) \setminus \{\text{current}(\boldsymbol{a})\} \\ &= \text{visited}(\boldsymbol{a}') \setminus \{\text{current}(\boldsymbol{a}')\} \\ &= \text{visited}(\text{parent}(\boldsymbol{a}')).\end{aligned}$$

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



				Direct					Via-depot				
Cost	Capacity	Visited	Current	0	1	2	3	4	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.

Fig. 4: Implementation of DPDP for VRP

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[15].

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 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 [16].

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*[17] 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

[15] 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.

[16] https://github.com/rusty1s/pytorch_scatter

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

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).

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.

2.3 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.[18]

2.4 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.

[18] 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.

References

1. Accorsi, L., Vigo, D.: A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems. Transportation Science **55**(4), 832–856 (2021)
2. Applegate, D., Bixby, R., Chvatal, V., Cook, W.: Concorde TSP solver (2006), http://www.math.uwaterloo.ca/tsp/concorde
3. Bai, R., Chen, X., Chen, Z.L., Cui, T., Gong, S., He, W., Jiang, X., Jin, H., Jin, J., Kendall, G., et al.: Analytics and machine learning in vehicle routing research. arXiv preprint arXiv:2102.10012 (2021)
4. Bellman, R.: On the theory of dynamic programming. Proceedings of the National Academy of Sciences of the United States of America **38**(8), 716 (1952)
5. Bellman, R.: Dynamic programming treatment of the travelling salesman problem. Journal of the ACM (JACM) **9**(1), 61–63 (1962)
6. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940 (2016)
7. Bertsekas, D.: Dynamic programming and optimal control: Volume I, vol. 1. Athena scientific (2017)
8. Cappart, Q., Moisan, T., Rousseau, L.M., Prémont-Schwarz, I., Cire, A.: Combining reinforcement learning and constraint programming for combinatorial optimization. AAAI Conference on Artificial Intelligence (AAAI) (2021)
9. Chen, X., Tian, Y.: Learning to perform local rewriting for combinatorial optimization. In: Advances in Neural Information Processing Systems (NeurIPS). pp. 6281–6292 (2019)
10. Cook, W., Seymour, P.: Tour merging via branch-decomposition. INFORMS Journal on Computing **15**(3), 233–248 (2003)
11. da Costa, P.R.d.O., Rhuggenaath, J., Zhang, Y., Akcay, A.: Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. Asian Conference on Machine Learning (ACML) (2020)
12. Da Silva, R.F., Urrutia, S.: A general vns heuristic for the traveling salesman problem with time windows. Discrete Optimization **7**(4), 203–211 (2010)
13. Daumé III, H., Marcu, D.: Learning as search optimization: Approximate large margin methods for structured prediction. In: International Conference on Machine Learning (ICML). pp. 169–176 (2005)
14. Delarue, A., Anderson, R., Tjandraatmadja, C.: Reinforcement learning with combinatorial actions: An application to vehicle routing. Advances in Neural Information Processing Systems (NeurIPS) **33** (2020)
15. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.M.: Learning heuristics for the TSP by policy gradient. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR). pp. 170–181. Springer (2018)
16. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
17. Dumas, Y., Desrosiers, J., Gelinas, E., Solomon, M.M.: An optimal algorithm for the traveling salesman problem with time windows. Operations Research **43**(2), 367–371 (1995)
18. Falkner, J.K., Schmidt-Thieme, L.: Learning to solve vehicle routing problems with time windows through joint attention. arXiv preprint arXiv:2006.09100 (2020)
19. Fu, Z.H., Qiu, K.B., Zha, H.: Generalize a small pre-trained model to arbitrarily large tsp instances. AAAI Conference on Artificial Intelligence (AAAI) (2021)

20. Gao, L., Chen, M., Chen, Q., Luo, G., Zhu, N., Liu, Z.: Learn to design the heuristics for vehicle routing problem. International Workshop on Heuristic Search in Industry (HSI) at the International Joint Conference on Artificial Intelligence (IJCAI) (2020)
21. Gasse, M., Chetelat, D., Ferroni, N., Charlin, L., Lodi, A.: Exact combinatorial optimization with graph convolutional neural networks. In: Advances in Neural Information Processing Systems (NeurIPS) (2019)
22. Gromicho, J., van Hoorn, J.J., Kok, A.L., Schutten, J.M.: Restricted dynamic programming: a flexible framework for solving realistic vrps. Computers & Operations Research **39**(5), 902–909 (2012)
23. Gromicho, J.A., Van Hoorn, J.J., Saldanha-da Gama, F., Timmer, G.T.: Solving the job-shop scheduling problem optimally by dynamic programming. Computers & Operations Research **39**(12), 2968–2977 (2012)
24. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021), https://www.gurobi.com
25. van Heeswijk, W., La Poutré, H.: Approximate dynamic programming with neural networks in linear discrete action spaces. arXiv preprint arXiv:1902.09855 (2019)
26. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied Mathematics **10**(1), 196–210 (1962)
27. Helsgaun, K.: An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems: Technical report (2017)
28. van Hoorn, J.J.: Dynamic Programming for Routing and Scheduling. Ph.D. thesis (2016)
29. Hottung, A., Bhandari, B., Tierney, K.: Learning a latent search space for routing problems using variational autoencoders. In: International Conference on Learning Representations (ICML) (2021)
30. Hottung, A., Tierney, K.: Neural large neighborhood search for the capacitated vehicle routing problem. European Conference on Artificial Intelligence (ECAI) (2020)
31. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: International Conference on Machine Learning (ICML). pp. 448–456 (2015)
32. Joshi, C.K., Laurent, T., Bresson, X.: An efficient graph convolutional network technique for the travelling salesman problem. INFORMS Annual Meeting (2019)
33. Joshi, C.K., Laurent, T., Bresson, X.: On learning paradigms for the travelling salesman problem. Graph Representation Learning Workshop at Neural Information Processing Systems (NeurIPS) (2019)
34. Kim, M., Park, J., Kim, J.: Learning collaborative policies to solve np-hard routing problems. In: Advances in Neural Information Processing Systems (NeurIPS) (2021)
35. Kok, A., Hans, E.W., Schutten, J.M., Zijm, W.H.: 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)
36. Kool, W., van Hoof, H., Welling, M.: Attention, learn to solve routing problems! In: International Conference on Learning Representations (ICLR) (2019)
37. Kwon, Y.D., Choo, J., Kim, B., Yoon, I., Gwon, Y., Min, S.: Pomo: Policy optimization with multiple optima for reinforcement learning. Advances in Neural Information Processing Systems (NeurIPS) (2020)

38. Laporte, G.: The vehicle routing problem: An overview of exact and approximate algorithms. European Journal of Operational Research (EJOR) **59**(3), 345–358 (1992)
39. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
40. Lee, J., Lee, Y., Kim, J., Kosiorek, A., Choi, S., Teh, Y.W.: Set transformer: A framework for attention-based permutation-invariant neural networks. In: International Conference on Machine Learning (ICML). pp. 3744–3753. PMLR (2019)
41. Li, S., Yan, Z., Wu, C.: Learning to delegate for large-scale vehicle routing. In: Advances in Neural Information Processing Systems (NeurIPS) (2021)
42. Li, Z., Chen, Q., Koltun, V.: Combinatorial optimization with graph convolutional networks and guided tree search. Advances in Neural Information Processing Systems (NeurIPS) p. 539 (2018)
43. Lu, H., Zhang, X., Yang, S.: A learning-based iterative method for solving vehicle routing problems. In: International Conference on Learning Representations (2020)
44. Ma, Q., Ge, S., He, D., Thaker, D., Drori, I.: Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. AAAI International Workshop on Deep Learning on Graphs: Methodologies and Applications (DLGMA) (2020)
45. Ma, Y., Li, J., Cao, Z., Song, W., Zhang, L., Chen, Z., Tang, J.: Learning to iteratively solve routing problems with dual-aspect collaborative transformer. In: Advances in Neural Information Processing Systems (NeurIPS) (2021)
46. Malandraki, C., Dial, R.B.: A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. European Journal of Operational Research (EJOR) **90**(1), 45–55 (1996)
47. Mazyavkina, N., Sviridov, S., Ivanov, S., Burnaev, E.: Reinforcement learning for combinatorial optimization: A survey. arXiv preprint arXiv:2003.03600 (2020)
48. Mingozzi, A., Bianco, L., Ricciardelli, S.: Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. Operations Research **45**(3), 365–377 (1997)
49. Nair, V., Bartunov, S., Gimeno, F., von Glehn, I., Lichocki, P., Lobov, I., O'Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., et al.: Solving mixed integer programs using neural networks. arXiv preprint arXiv:2012.13349 (2020)
50. Nazari, M., Oroojlooy, A., Snyder, L., Takac, M.: Reinforcement learning for solving the vehicle routing problem. In: Advances in Neural Information Processing Systems (NeurIPS). pp. 9860–9870 (2018)
51. Novoa, C., Storer, R.: An approximate dynamic programming approach for the vehicle routing problem with stochastic demands. European Journal of Operational Research (EJOR) **196**(2), 509–515 (2009)
52. Nowak, A., Villar, S., Bandeira, A.S., Bruna, J.: A note on learning algorithms for quadratic assignment with graph neural networks. In: Principled Approaches to Deep Learning Workshop at the International Conference on Machine Learning (ICML) (2017)
53. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in Neural Information Processing Systems (NeurIPS) **32**, 8026–8037 (2019)
54. Peng, B., Wang, J., Zhang, Z.: A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. In: International Symposium on Intelligence Computation and Applications. pp. 636–650. Springer (2019)

55. Ropke, S., Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Science **40**(4), 455–472 (2006)
56. Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., Dueck, G.: Record breaking optimization results using the ruin and recreate principle. Journal of Computational Physics **159**(2), 139–171 (2000)
57. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (2018)
58. Sun, Y., Ernst, A., Li, X., Weiner, J.: Generalization of machine learning for problem reduction: a case study on travelling salesman problems. OR Spectrum pp. 1–27 (2020)
59. Toth, P., Vigo, D.: Vehicle routing: problems, methods, and applications. SIAM (2014)
60. Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., Subramanian, A.: New benchmark instances for the capacitated vehicle routing problem. European Journal of Operational Research (EJOR) **257**(3), 845–858 (2017)
61. Vesselinova, N., Steinert, R., Perez-Ramirez, D.F., Boman, M.: Learning combinatorial optimization on graphs: A survey with applications to networking. IEEE Access **8**, 120388–120416 (2020)
62. Vidal, T.: Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. arXiv preprint arXiv:2012.10384 (2020)
63. Vidal, T., Crainic, T.G., Gendreau, M., Lahrichi, N., Rei, W.: A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. Operations Research **60**(3), 611–624 (2012)
64. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in Neural Information Processing Systems (NeurIPS). pp. 2692–2700 (2015)
65. Wiseman, S., Rush, A.M.: Sequence-to-sequence learning as beam-search optimization. In: Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1296–1306 (2016)
66. Wu, Y., Song, W., Cao, Z., Zhang, J., Lim, A.: Learning improvement heuristics for solving routing problems. IEEE Transactions on Neural Networks and Learning Systems (2021)
67. Xin, L., Song, W., Cao, Z., Zhang, J.: Step-wise deep learning models for solving routing problems. IEEE Transactions on Industrial Informatics (2020)
68. Xin, L., Song, W., Cao, Z., Zhang, J.: NeuroLKH: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem. In: Advances in Neural Information Processing Systems (NeurIPS) (2021)
69. Xu, S., Panwar, S.S., Kodialam, M., Lakshman, T.: Deep neural network approximated dynamic programming for combinatorial optimization. In: AAAI Conference on Artificial Intelligence (AAAI). vol. 34, pp. 1684–1691 (2020)
70. Yang, F., Jin, T., Liu, T.Y., Sun, X., Zhang, J.: Boosting dynamic programming with neural networks for solving np-hard problems. In: Asian Conference on Machine Learning (ACML). pp. 726–739. PMLR (2018)