Deep Policy Dynamic Programming for Vehicle Routing Problems

Wouter Kool [1 2] **Herke van Hoof** [1] **Joaquim Gromicho** [1 2] **Max Welling** [1 3]

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 can find 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) and the vehicle routing problem (VRP) 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 other 'neural approaches' for solving TSPs and VRPs 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 (Bellman, 1952). Famous examples are Dijkstra's algorithm (Dijkstra, 1959) for finding the shortest route between two locations, and the classic Held-Karp algorithm for the travelling salesman problem (TSP) (Held & Karp, 1962; Bellman, 1962). 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 (Vinyals et al., 2015; Bello et al., 2016; Nazari et al., 2018; Kool et al., 2019; Chen & Tian, 2019). While the results are promising, the learned heuristics are not (yet) competitive to 'traditional' algorithms such as LKH (Helsgaun, 2017) 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. DPDP uses a (deep) graph neural network (GNN) proposed in Joshi et al. (2019a) to preprocess a problem instance into a (sparse) heatmap of promising edges, from which a policy is derived which is used to guide a *restricted dynamic programming* algorithm (Gromicho et al., 2012a) to construct a feasible solution. Related ideas have been proposed by Yang et al. (2018); van Heeswijk & La Poutré (2019); Xu et al. (2020) but DPDP is more powerful since it combines simple supervised training with DP at test time.

The key of DPDP is to combine the strengths of deep learning with dynamic programming, by restricting the dynamic programming state space using the policy derived from the neural network. This allows the neural network to direct the dynamic program towards promising regions in the solution space, where the dynamic programming algorithm is powerful at finding the best solutions in those regions. Intuitively, the neural network 'sketches' the outlines of a solution (through the heatmap) and the dynamic program fills in the details (i.e. the exact order to visit nodes) optimally. This is especially interesting as the dynamic programming framework is very flexible to model realistic routing problems with difficult practical constraints (Gromicho et al., 2012a).

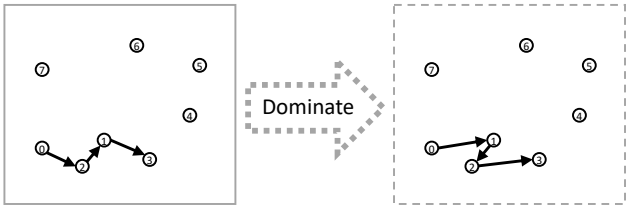

Figure 1. Two partial solutions for TSP, corresponding to the same DP state (nodes visited and current node). The right partial solution is dominated by the left since any completion of the right solution is longer than the same completion of the left solution.

[1]University of Amsterdam, The Netherlands [2]ORTEC, The Netherlands [3]CIFAR, Canada. Correspondence to: Wouter Kool <w.w.m.kool@uva.nl>.

Preliminary work.

Our DPDP is a forward iterative restricted dynamic programming algorithm, which starts from an empty solution and expands at most the B highest-scoring partial solutions (as defined by the policy) in each iteration. In each iteration, DPDP applies the dynamic programming principle and keeps only the best solution(s) for each DP state by removing dominated solutions (see Figure 1). The resulting algorithm is a *beam search* over the DP state space, which gets restricted by the *beam size* B and the policy for selecting the B solutions to expand, which we refer to as the *beam*. DPDP is asymptotically optimal as choosing $B = n \cdot 2^n$ for a TSP with n nodes guarantees optimal results (in a fully connected graph). The parameter B allows to trade off the performance and the computational cost of the algorithm.

The *policy* for choosing the beam is important as it defines the part of the DP state space that is considered and therefore the quality of the solution found. Typical examples are using the *cost* for each partial solution, which makes the DP 'greedy' (but with a wider view) or the cost plus some (heuristic) estimate of the *cost-to-go*: the (optimal) cost to complete the partial solution. A good estimate of the cost-to-go will yield better results with smaller beam sizes, but may be computationally expensive to obtain (van Hoorn, 2016). An oracle providing the true cost-to-go renders a beam size of 1 optimal. In practice, one needs to trade off the quality of the estimate and its computational cost.

In this work, we leverage recent advances in deep learning and use a graph neural network to define a score for each partial solution, implicitly defining a policy to select the B solutions in the beam. Prior work on 'neural' vehicle routing has focused on auto-regressive models (Vinyals et al., 2015; Bello et al., 2016; Deudon et al., 2018; Kool et al., 2019), but they have high computational cost in combination with DP, as the model needs to be evaluated for each partial solution at each iteration. Instead, we use (for TSP) and adapt (for VRP) the model by Joshi et al. (2019a) to predict a heatmap indicating promising edges, and define the score as the 'heat' of the edges in the current partial solution plus an estimate of the 'heat-to-go', which we refer to as the *potential* of the solution. This enables a much larger beam size as the neural network only needs to be evaluated *once* for each instance. 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 ruling out some solutions.

Figure 2 illustrates the overall DPDP algorithm. In Section 4, we show that DPDP significantly improves the 'classic' DP algorithm, when restricted to the same beam size. Additionally, we show that DPDP outperformes other 'neural' approaches for TSP and VRP and is competitive with the highly-optimized LKH solver (Helsgaun, 2017) for VRP, on two different data distributions, including the more realistic and challenging data distribution by Uchoa et al. (2017).

2. Related work

2.1. Dynamic Programming for Vehicle Routing Problems

DP has a long history as an exact solution method for routing problems (see, e.g. Laporte (1992); Toth & Vigo (2014)), e.g. for the TSP with time windows (Dumas et al., 1995) and precedence constraints (Mingozzi et al., 1997). However, because of the curse of dimensionality, the application of (unrestricted) DP is limited to small problems only. Malandraki & Dial (1996) considered the time dependent TSP, which is challenging for (non-)linear programming and heuristics, and showed how a restricted dynamic programming heuristic can be applied. This method has been generalized by Gromicho et al. (2012a) as a flexible framework for VRPs with different types of practical constraints. DP approaches have also been shown to be useful in settings with difficult practical issues such as time-dependent travel times and driving regulations (Kok et al., 2010) or stochastic demands (Novoa & Storer, 2009). For a thorough investigation of modelling choices of DP for routing (and scheduling), see van Hoorn (2016). For sparse graphs, alternative formulations can be used (e.g. Cook & Seymour (2003)) but these are less flexible.

Despite the flexibility, DP methods have not gained much popularity compared to highly heuristic approaches such as Ruin and Recreate (Schrimpf et al., 2000), Adaptive Large Neighborhood Search (Ropke & Pisinger, 2006), LKH (Helsgaun, 2017) or FILO (Accorsi & Vigo, 2020). While highly effective, these methods are limited in their flexibility as special operators need to be engineered for different types of problems. While restricted DP was shown to have superior performance on *realistic* VRPs with many constraints (Gromicho et al., 2012a), the performance gap of around 10% for standard (benchmark) VRPs (with time windows) is too large to popularize the restricted dynamic programming approach. We argue that the missing ingredient for restricted dynamic programming is the availability of a strong but computationally cheap policy for selecting which solutions should be considered, which is the motivation behind DPDP, which shows good performance for the standard capacitated VRP (see Section 4).

2.2. Machine Learning for Vehicle Routing Problems

In the machine learning community, recent advances in methods and hardware have significantly improved the performance of deep neural networks (DNNs) to perform non-trivial tasks such as image classification, machine translation and many more (LeCun et al., 2015). Vinyals et al. (2015) were the first to apply 'modern deep learning' to the TSP, by training a sequence model to construct TSP tours, using a training set of example solutions. From there, many improvements have been proposed, e.g. different training

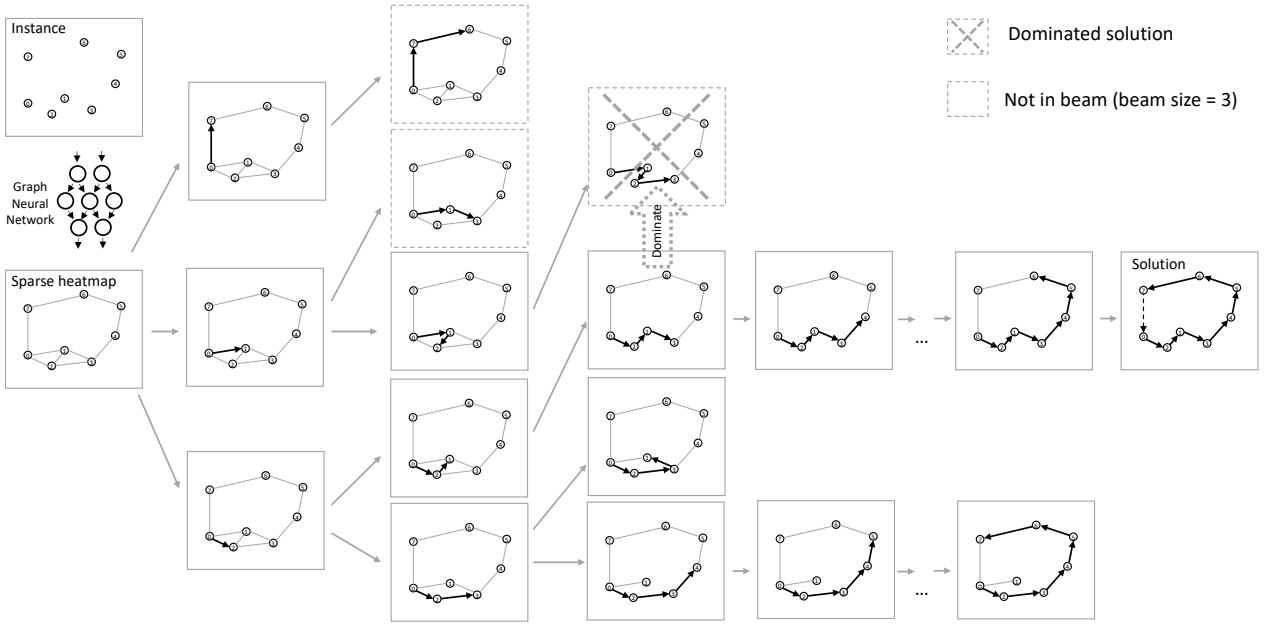

Figure 2. Deep Policy Dynamic Programming for the TSP with a beam size of 3. A GNN creates a (sparse) heatmap with promising edges, after which a tour is constructed using forward dynamic programming. In each step, at most $B = 3$ solutions are expanded. In step 3 it can be seen how one of the partial solutions is dominated by a shorter (lower cost) solution with the same nodes visited and current node.

strategies such as reinforcement learning (Bello et al., 2016; Joshi et al., 2019b; Delarue et al., 2020) and model architectures, which have enabled the same idea to be used for other routing problems (Nazari et al., 2018; Kool et al., 2019; Deudon et al., 2018; Peng et al., 2019; Falkner & Schmidt-Thieme, 2020; Xin et al., 2020).

Most constructive neural methods are *auto-regressive*, predicting the next node given the partial tour constructed, but other works have considered predicting a 'heatmap' of promising edges *at once* (Nowak et al., 2017; Joshi et al., 2019a; Fu et al., 2020), which allows a tour to be constructed (using sampling or beam search) without further evaluating the model. Whereas these are constructive approaches, others have reported results with 'learning to search', where a neural network is used to guide a search procedure such as local search (Chen & Tian, 2019; Lu et al., 2020; Gao et al., 2020; Wu et al., 2019; Hottung & Tierney, 2019). While most researches have focused on instances up to 100 nodes, some have attempted scaling to larger instances, which remains challenging (Ma et al., 2019; Fu et al., 2020). Related to our approach, Cappart et al. (2020) propose to combine reinforcement learning, constraint programming and dynamic programming and experiment with the TSP with time windows. For surveys of machine learning for routing problems and combinatorial optimization in general, we refer to Mazyavkina et al. (2020); Vesselinova et al. (2020).

3. Deep Policy Dynamic Programming

DPDP uses the graph neural network from Joshi et al. (2019a) (which we modify for VRP) 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 then starts with a beam of a single initial (empty) solution and in iterations expands all solutions on the beam, then removes dominated solutions and finally selects the B best solutions according to the policy to define the beam for the next iteration. This is a generic framework that can be applied to different problems, by defining the following ingredients:

- The state variables to track while constructing solutions
- The initial solution
- Feasible actions to expand solutions
- Rules that define which solutions should be dominated
- A scoring policy for selecting the B solutions to keep

A solution should always be (uniquely) defined by the sequence of actions. This allows the DP algorithm to construct the final solution by backtracking the actions that have been used to expand the solutions. In the next sections, we define the ingredients for the TSP and the VRP.

3.1. Travelling Salesman Problem

We implement our DP algorithm for Euclidean TSPs with n nodes on a (sparse) graph, where the cost (Euclidean distance) for edge (i, j) between nodes i and j is given by c_{ij} (but we do assume node coordinates are available as input for the neural network model). We experiment with different strategies for defining the sparsity structure (see Section 4.3.3).

3.1.1. State variables

For each partial solution, defined by a sequence of actions $\boldsymbol{a}$, we keep track of $\text{cost}(\boldsymbol{a})$, 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).

3.1.2. Initial solution

Without loss of generality, we let 0 be the start node, so we initialize the beam at step $t = 0$ with the empty solution with $\text{cost}(\boldsymbol{a}) = 0$, $\text{current}(\boldsymbol{a}) = 0$ and $\text{visited}(\boldsymbol{a}) = \{0\}$.

3.1.3. Actions

At step t the action $a_t \in \{0, ..., n-1\}$ indicates the next node to visit. An action a_t defines a feasible expansion 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 state variables incrementally:

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

$$\text{current}(\boldsymbol{a}') = a_t, \tag{2}$$

$$\text{visited}(\boldsymbol{a}') = \text{visited}(\boldsymbol{a}) \cup \{a_t\}. \tag{3}$$

3.1.4. Dominated solutions

A (partial) solution $\boldsymbol{a}$ dominates another partial solution $\boldsymbol{a}'$ if $\text{visited}(\boldsymbol{a}) = \text{visited}(\boldsymbol{a}')$ and $\text{current}(\boldsymbol{a}) = \text{current}(\boldsymbol{a}')$ and $\text{cost}(\boldsymbol{a}) < \text{cost}(\boldsymbol{a}')$. The tuple $(\text{visited}(\boldsymbol{a}), \text{current}(\boldsymbol{a}))$ we refer to as the *DP state*, so removing all dominated partial solutions, we keep exactly one minimum-cost solution for each unique DP state[1].

We refer to the action of removing dominated (partial) solutions from a set of solutions as *collapsing*. Since a solution can only dominate other solutions with the same set of visited nodes, we only need to collapse sets of solutions with the same number of actions. Therefore, we can execute the DP algorithm in iterations, where at step t all solutions have t actions (and $t + 1$ visited nodes, including the start node).

[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.

3.1.5. The scoring policy

We use a pretrained model from Joshi et al. (2019a), 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

$$\text{score}(\boldsymbol{a}) = \text{heat}(\boldsymbol{a}) + \text{potential}(\boldsymbol{a}). \tag{4}$$

The heat of the solution $\boldsymbol{a}$ is the sum of the heat of the edges:

$$\text{heat}(\boldsymbol{a}) = \sum_{i=1}^{t-1} h_{a_{i-1},a_i}. \tag{5}$$

Note that this can be computed incrementally when expanding a solution. We could simply use the heat as score, but this may result in some nearby nodes being 'skipped' if other edges have higher heatmap values. The skipped nodes must be visited later, resulting in bad or 'dead-end' solutions (with a sparse graph), similar to node 1 being skipped in the bottom row in Figure 2. To avoid this 'greedy pitfall', we also take into account the heat *potential* (for incoming edges) for the remaining unvisited nodes (and the start node). The heat potential is defined as

$$\text{potential}(\boldsymbol{a}) = \text{potential}_0(\boldsymbol{a}) + \sum_{i \notin \text{visited}(\boldsymbol{a})} \text{potential}_i(\boldsymbol{a}), \tag{6}$$

where $\text{potential}_i(\boldsymbol{a})$ is the remaining heat potential for node i, which can be computed incrementally and is defined as

$$\text{potential}_i(\boldsymbol{a}) = w_i \sum_{j \notin \text{visited}(\boldsymbol{a})} \frac{h_{ji}}{\sum_{j'=0}^{n-1} h_{j'i}}, \tag{7}$$

where w_i is the node *potential weight* given by

$$w_i = \left(\max_j h_{ji}\right) \cdot \left(1 - 0.1\left(\frac{c_{i0}}{\max_j c_{j0}} - 0.5\right)\right). \tag{8}$$

By normalizing the heatmap values for incoming edges in equation (7), 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 (as this node cannot contribute more to the heat than this value), which we multiply by a factor 0.95 to 1.05 to give a higher weight to nodes closer to the start node (see Equation (8)). This is a small inductive bias which we found helps to encourage the algorithm to keep edges that enable to return to the start node. The overall score in Equation (4) is designed to identify promising partial solutions based on the heatmap values.

3.2. Vehicle Routing Problem

For the VRP, we add a special depot node to the graph of the problem instance, which we indicate using the special token DEP. Each node i has a demand d_i, and the goal is to find multiple routes, where each route has a limited vehicle capacity, which we denote by CAPACITY. In this section we explain how we adapt the DP formulation and we describe how to adapt the model by Joshi et al. (2019a) and train it on example VRP solutions to predict the heatmap values for the scoring policy.

3.2.1. STATE VARIABLES

Additionally to the state 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.

3.2.2. INITIAL SOLUTION

The solution starts at the depot, so we initialize the beam at step $t = 0$ with the empty solution with $\text{cost}(\boldsymbol{a}) = 0$, $\text{current}(\boldsymbol{a}) = \text{DEP}$, $\text{visited}(\boldsymbol{a}) = \emptyset$ and $\text{capacity}(\boldsymbol{a}) = \text{CAPACITY}$.

3.2.3. ACTIONS

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*. For the first action a_0 there is no choice and we constrain (for convenience of implementation) $a_0 \in \{n, ..., 2n-1\}$. Expanding a solution $\boldsymbol{a}$ via the depot is always feasible (assuming $d_i \leq \text{CAPACITY}$ and respecting the graph edges), whereas a direct move ($a_t < n$) is only feasible if $d_{a_t} \leq \text{capacity}(\boldsymbol{a})$. 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 similar to the TSP formulation: at step t we have exactly t nodes visited, and we can collapse solutions for only step t.

It is helpful to define the cost of edge (i, j) *via the depot*:

$$c_{ij}^{\text{DEP}} = c_{i,\text{DEP}} + c_{\text{DEP},j} \tag{9}$$

so we can incrementally compute the state variables as

$$\text{cost}(\boldsymbol{a}') = \text{cost}(\boldsymbol{a}) + \begin{cases} c_{\text{current}(a),a_t} & \text{if } a_t < n \\ c^{\text{DEP}}_{\text{current}(a),a_t-n} & \text{if } a_t \geq n \end{cases} \tag{10}$$

$$\text{current}(\boldsymbol{a}') = a_t \bmod n \tag{11}$$

$$\text{visited}(\boldsymbol{a}') = \text{visited}(\boldsymbol{a}) \cup \{a_t \bmod n\} \tag{12}$$

$$\text{capacity}(\boldsymbol{a}') = \begin{cases} \text{capacity}(\boldsymbol{a}) - d_{a_t} & \text{if } a_t < n \\ \text{CAPACITY} - d_{a_t-n} & \text{if } a_t \geq n \end{cases}. \tag{13}$$

3.2.4. DOMINATED SOLUTIONS

For VRP, a partial solution $\boldsymbol{a}$ dominates another partial solution $\boldsymbol{a}'$ if $\text{visited}(\boldsymbol{a}) = \text{visited}(\boldsymbol{a}')$ and $\text{current}(\boldsymbol{a}) = \text{current}(\boldsymbol{a}')$ (i.e. they correspond 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 S and current node i, 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 solution may still be desired if the remaining vehicle capacity is also higher, and vice versa.

3.2.5. THE SCORING POLICY

For the VRP, we modify the model by Joshi et al. (2019a) to include the depot node and demands. The special depot node gets a learned initial embedding parameter different from 'normal' nodes, and we add additional edge types for connections to the depot, to enable the model to recognize the depot as being special. Additionally, each node gets an additional input (next to the two coordinates) corresponding to $d_i/\text{CAPACITY}$ (where we set the demand for the depot equal to 0). Apart from this, the model remains exactly the same[2]. The model is trained on example solutions solved using LKH (Helsgaun, 2017) (see Section 4.2). Contrasting the TSP dataset used for training, these solutions are not optimal, but still provide a useful training signal, which highlights the flexibility of the approach.

The definition of the heat is slightly changed to accommodate for the via-depot actions and is best defined incrementally. Similar to Equations (9) and (10) for the cost, we define the heatmap value via the depot:

$$h_{ij}^{\text{DEP}} = h_{i,\text{DEP}} \cdot h_{\text{DEP},j} \cdot 0.1. \tag{14}$$

We multiply the values rather than add them to keep the heatmap values in the range $(0, 1)$ and multiply by an additional penalty factor of 0.1 for visiting the depot, to encourage 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 computed incrementally as:

$$\text{heat}(\boldsymbol{a}') = \text{heat}(\boldsymbol{a}) + \begin{cases} h_{\text{current}(\boldsymbol{a}),a_t} & \text{if } a_t < n \\ h^{\text{DEP}}_{\text{current}(\boldsymbol{a}),a_t-n} & \text{if } a_t \geq n \end{cases}. \tag{15}$$

The potential is defined identically to the TSP, but we replace the start node 0 by the special depot node DEP in Equations (6) and (8).

[2]Except that we do not use the K-nearest neighbour indicator feature as described by Joshi et al. (2019a) as it contains no additional information.

3.3. Graph sparsity

As described, the DP algorithm can take into account a sparse graph structure when considering feasible expansions. As the problems considered 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* (Sun et al., 2020). For the KNN graph, we add edges in both directions to make the graph symmetric, and for the VRP we additionally connect each node to the depot (and vice versa) to ensure feasibility.

3.4. Implementation & hyperparameters

We implement DPDP using PyTorch (Paszke et al., 2017) to leverage batched computation on the GPU. For details, see Appendix A. Our code is publicly available.[3] DPDP has very few hyperparameters, but the heatmap threshold of 10^{-5} and some 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 manually selected using 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 the test set by Kool et al. (2019). We report results using Concorde (Applegate et al., 2006), LKH (Helsgaun, 2017) and Gurobi (Gurobi Optimization, LLC, 2018), as well as recent results from literature using the strongest 'neural approaches'. Running times should be taken as rough indications as they are on different machines, typically with 1 GPU or a many-core CPU (8 - 32). The values reported by Fu et al. (2020) are slightly different from ours, making the cost value not directly comparable. Similar to Fu et al. (2020) we report the time for generating the heatmaps seperately from the running time for the DP algorithm. DPDP achieves close to optimal results, outperforming all neural baselines, although somewhat slower since our DPDP implementation is not optimized for small beam sizes[4].

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

[4] 10K should be 10× faster than 100K, i.e. ± 15 minutes

Table 1. Main results for TSP with 100 nodes.

METHOD	COST	GAP	TIME
CONCORDE	7.765	0.000 %	6M
LKH	7.765	0.000 %	42M
GUROBI	7.776	0.15 %	31M
KOOL ET AL. (2019)	7.94	2.26 %	1H
JOSHI ET AL. (2019A)	7.87	1.39 %	40M
DA COSTA ET AL. (2020)	7.83	0.87 %	41M
FU ET AL. (2020)	7.764*	0.04 %	4M + 11M
DPDP 10K	7.765	0.009 %	10M + 1H06M
DPDP 100K	7.765	0.004 %	10M + 2H35M

Table 2. Main results for VRP with 100 nodes.

METHOD	COST	GAP	TIME
LKH	15.647	0.000 %	12H59M
XIN ET AL. (2020)	16.49	4.99 %	39S
KOOL ET AL. (2019)	16.23	3.72 %	2H
CHEN & TIAN (2019)	16.10	2.90 %	1H
PENG ET AL. (2019)	16.27	3.96 %	6H
WU ET AL. (2019)	16.03	2.47 %	5H
HOTTUNG & TIERNEY (2019)	15.99	1.02 %	1H
LU ET AL. (2020)	15.57*	-	4000H
DPDP 10K	15.832	1.183 %	10M + 2H24M
DPDP 100K	15.694	0.305 %	10M + 5H48M
DPDP 1M	15.627	- 0.127 %	10M + 48H27M

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 Nazari et al. (2018) and solved using one run of LKH (Helsgaun, 2017). 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 Joshi et al. (2019a)) from which we select the saved checkpoint with the lowest validation loss. We use the validation and test sets by Kool et al. (2019).

Table 2 shows the results for LKH, the strongest neural approaches and DPDP with beam sizes up to 1 million. Results of Lu et al. (2020) are for 2000 instances and cannot be directly compared[5]. DPDP outperforms all other neural baselines and is competitive to LKH (see also Section 4.3.2).

[5] The running time of 4000 hours (167 days) for 10K instances is estimated from their reported avg. runtime of 24min/instance.

Table 3. Main results for VRP with 100 nodes on 10000 instances generated using the data distribution by Uchoa et al. (2017)

METHOD	COST	GAP	TIME 1 GPU OR 16 CPUS	TIME 4 GPUS OR 32 CPUS
LKH	18133	0.000 %	25H59M	13H
DPDP 10K	18415	1.550 %	10M + 2H24M	2M + 36M
DPDP 100K	18253	0.657 %	10M + 5H48M	2M + 1H27M
DPDP 1M	18168	0.191 %	10M + 48H27M	2M + 12H7M

4.2.1. MORE REALISTIC INSTANCES

We also train the model and run experiments with instances with 100 nodes from the more realistic and challenging data distribution by Uchoa et al. (2017). This distribution, commonly used in the routing community, has greater variability, in terms of node clustering and demand distributions. LKH takes about twice as long to solve these instances, so for practical reasons we only solve 250K instances as training set and use 10K instances for the validation and test set. All other hyperparameters are the same. LKH failed to solve two of the test instances, which we found out is because LKH by default uses a fixed number of vehicles 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[6]. Therefore we solve these instances by rerunning LKH with an unlimited number of allowed vehicles (which in general gives worse results, see Section 4.3.2).

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 LKH. In Table 3 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 LKH is impressive for these challenging instances, and we consider the runtime (for solving 10K instances) acceptable, especially when using multiple GPUs.

4.3. Ablations

4.3.1. SCORING POLICY

To evaluate the value of the used scoring policy, as well as the dynamic programming principle, we run three variants of DP, as well as a standard beam search (BS) using different beam sizes up to 100K, on 100 validation instances:

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

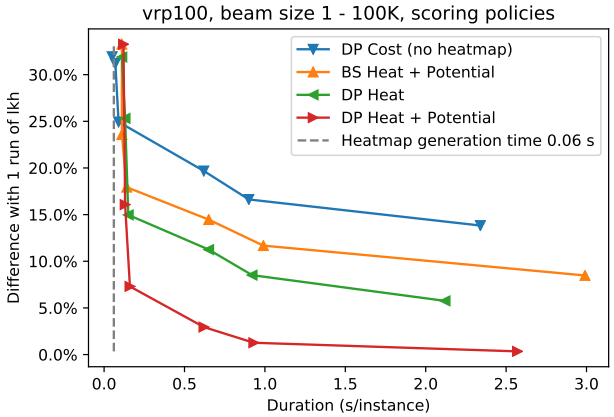

Figure 3. DPDP with different score functions, as well as a 'pure' beam search version which does not remove dominated solutions. Results are reported for beam sizes 1, 10, 100, 1000, 10K, 100K.

- **DP Cost**, which selects the beam based on the current cost of the solution. This does not use the neural network so is as 'classic' instance of (restricted) DP.
- **BS Heat + Potential**, which runs a 'pure' beam search which does not remove dominated solutions, but does select the beam based on the heat and the potential.
- **DP Heat**, which does not add the potential.
- **DP Heat + Potential**, which is the full DPDP version.

Each variant uses the fully connected graph (knn $= n - 1$), such that the effect of the predictions from the neural network is only through the scoring policy. In Figure 3 it can be seen that DP using the heat without potential is significantly worse than with the potential, but still significantly better than 'classic' DP using the current cost. Also, it is clear that using DP significantly improves over a standard beam search by removing dominated solutions. The figure also illustrates how the heatmap generation time using the neural network only makes up a small portion of the total runtime.

4.3.2. 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 4.3.3). We illustrate this trade-off in Figure 4, where we evaluate DPDP on 100 validation instances, with different beam sizes from 10K to 2.5M. We also report the trade-off curve for the strongest baseline LKH(U), where 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 4.2.1). It is hard to compare GPU vs CPU, so we report (estimated) runtimes

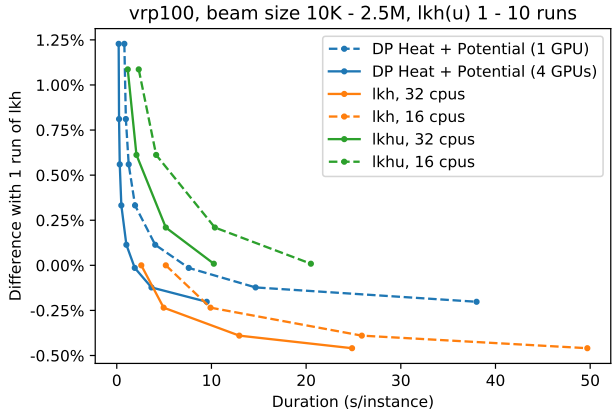

Figure 4. DPDP with beam sizes 10K, 25K, 50K, 100K, 250K, 500K, 1M, 2.5M compared against two variants of LKH, reporting the best of 1, 2, 5 and 10 runs.

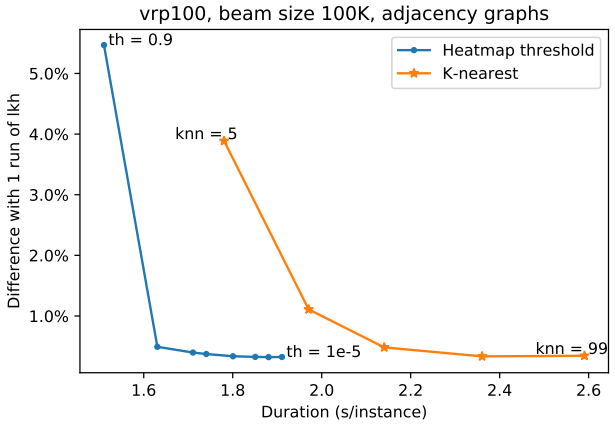

Figure 5. DPDP with different sparse adjacency graphs, defined by heatmap thresholds 0.9, 0.5, 0.2, 0.1, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5} and K-nearest neighbours with knn $= 5, 10, 20, 50, 99$. Results are reported for VRP with 100 nodes and a beam size of 100K.

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 a single run of LKH, which corresponds to the value for LKH reported in Table 2 and in related work, e.g. Kool et al. (2019). We thus compare against a stronger baseline by allowing LKH more runs. We stress however that the goal is not to outperform LKH, but to show DPDP has reasonable performance compared to a highly optimized solver while being promising as a flexible framework for other (routing) problems.

4.3.3. Graph sparsity

We test the two graph sparsification strategies described in Section 3.3 as another way to trade off performance and runtime of DPDP. In Figure 5, 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 much sparser graphs (and thus lower runtimes). This illustrates that the heatmap threshold strategy is much more informed than the KNN strategy, confirming the significant 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 solver for VRPs with 100 nodes. The method has great potential for further improvements, as dynamic programming is a flexible framework (illustrated by results on both TSP and VRP whereas many baselines focus on one) and the supervised training pipeline can be adapted to new problems, given that a (heuristic) solver is available to provide example solutions.

We see many future directions of this work. Scaling to a larger number of nodes is challenging, especially for training the model, but possible opportunities are using a sparse graph neural networks (e.g. KNN or a learned sparsification) for better scaling than fully connected $O(n^2)$ models. Other directions are to train the scoring policy end-to-end, which is now manually defined as a function of the heatmap. Another interesting direction would be to train the model 'tabula rasa', as the DP algorithm can be seen analogously to MCTS used in *AlphaZero* (Silver et al., 2018).

We think DPDP has high potential for solving other, more constrained, routing problems, such as the VRP with time windows. These are challenging for local search based approaches, as it is hard to maintain feasibility while modifying existing solutions. Dynamic programming is a constructive procedure and can more effectively check and incorporate complex constraints such as time windows. DP can also be applied to other problems such as job shop scheduling (Gromicho et al., 2012b), and we are interested to see if DPDP can bring significant improvements there as well.

In this work we tried to bring machine learning research for combinatorial optimization closer to the operations research (especially vehicle routing) community, by combining machine learning with dynamic programming and evaluating the new framework on both the data distributions by Nazari et al. (2018), commonly used in the machine learning community, and the more realistic distribution from Uchoa et al. (2017), commonly used in the vehicle routing community.

References

Accorsi, L. and Vigo, D. A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems. Technical report, Tech. rep., University of Bologna, 2020.

Applegate, D., Bixby, R., Chvatal, V., and Cook, W. Concorde TSP solver, 2006.

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.

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

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Cappart, Q., Moisan, T., Rousseau, L.-M., Prémont-Schwarz, I., and Cire, A. Combining reinforcement learning and constraint programming for combinatorial optimization. *arXiv preprint arXiv:2006.01610*, 2020.

Chen, X. and Tian, Y. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, volume 32, pp. 6281–6292, 2019.

Cook, W. and Seymour, P. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3): 233–248, 2003.

da Costa, P. R. d. O., Rhuggenaath, J., Zhang, Y., and Akcay, A. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. *Proceedings of Machine Learning Research*, 1:17, 2020.

Delarue, A., Anderson, R., and Tjandraatmadja, C. Reinforcement learning with combinatorial actions: An application to vehicle routing. *Advances in Neural Information Processing Systems*, 33, 2020.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., and 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*, pp. 170–181. Springer, 2018.

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

Dumas, Y., Desrosiers, J., Gelinas, E., and Solomon, M. M. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2):367–371, 1995.

Falkner, J. K. and Schmidt-Thieme, L. Learning to solve vehicle routing problems with time windows through joint attention. *arXiv preprint arXiv:2006.09100*, 2020.

Fu, Z.-H., Qiu, K.-B., and Zha, H. Generalize a small pre-trained model to arbitrarily large tsp instances. *arXiv preprint arXiv:2012.10658*, 2020.

Gao, L., Chen, M., Chen, Q., Luo, G., Zhu, N., and Liu, Z. Learn to design the heuristics for vehicle routing problem. *arXiv preprint arXiv:2002.08539*, 2020.

Gromicho, J., van Hoorn, J. J., Kok, A. L., and Schutten, J. M. Restricted dynamic programming: a flexible framework for solving realistic vrps. *Computers & operations research*, 39(5):902–909, 2012a.

Gromicho, J. A., Van Hoorn, J. J., Saldanha-da Gama, F., and Timmer, G. T. Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research*, 39(12):2968–2977, 2012b.

Gurobi Optimization, LLC. Gurobi, 2018.

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

Helsgaun, K. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems: Technical report. 2017.

Hottung, A. and Tierney, K. Neural large neighborhood search for the capacitated vehicle routing problem. *arXiv preprint arXiv:1911.09539*, 2019.

Joshi, C. K., Laurent, T., and Bresson, X. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*, 2019a.

Joshi, C. K., Laurent, T., and Bresson, X. On learning paradigms for the travelling salesman problem. *arXiv preprint arXiv:1910.07210*, 2019b.

Kok, A., Hans, E. W., Schutten, J. M., and 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.

Kool, W., van Hoof, H., and Welling, M. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

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

LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature*, 521(7553):436, 2015.

Lu, H., Zhang, X., and Yang, S. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2020.

Ma, Q., Ge, S., He, D., Thaker, D., and Drori, I. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *arXiv preprint arXiv:1911.04936*, 2019.

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

Mazyavkina, N., Sviridov, S., Ivanov, S., and Burnaev, E. Reinforcement learning for combinatorial optimization: A survey. *arXiv preprint arXiv:2003.03600*, 2020.

Mingozzi, A., Bianco, L., and Ricciardelli, S. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. *Operations research*, 45(3):365–377, 1997.

Nazari, M., Oroojlooy, A., Snyder, L., and Takac, M. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pp. 9860–9870, 2018.

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

Nowak, A., Villar, S., Bandeira, A. S., and Bruna, J. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint arXiv:1706.07450*, 2017.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Peng, B., Wang, J., and 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.

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

Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., and Dueck, G. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

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.

Sun, Y., Ernst, A., Li, X., and Weiner, J. Generalization of machine learning for problem reduction: a case study on travelling salesman problems. *OR Spectrum*, pp. 1–27, 2020.

Toth, P. and Vigo, D. *Vehicle routing: problems, methods, and applications*. SIAM, 2014.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., and Subramanian, A. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017.

van Heeswijk, W. and La Poutré, H. Approximate dynamic programming with neural networks in linear discrete action spaces. *arXiv preprint arXiv:1902.09855*, 2019.

van Hoorn, J. J. *Dynamic Programming for Routing and Scheduling*. PhD thesis, 2016.

Vesselinova, N., Steinert, R., Perez-Ramirez, D. F., and Boman, M. Learning combinatorial optimization on graphs: A survey with applications to networking. *IEEE Access*, 8:120388–120416, 2020.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Wu, Y., Song, W., Cao, Z., Zhang, J., and Lim, A. Learning improvement heuristics for solving routing problems. *arXiv preprint arXiv:1912.05784*, 2019.

Xin, L., Song, W., Cao, Z., and Zhang, J. Step-wise deep learning models for solving routing problems. *IEEE Transactions on Industrial Informatics*, 2020.

Xu, S., Panwar, S. S., Kodialam, M., and Lakshman, T. Deep neural network approximated dynamic programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 1684–1691, 2020.

Yang, F., Jin, T., Liu, T.-Y., Sun, X., and Zhang, J. Boosting dynamic programming with neural networks for solving np-hard problems. In *Asian Conference on Machine Learning*, pp. 726–739. PMLR, 2018.

A. Implementation

We implement the dynamic programming algorithm on the GPU using PyTorch (Paszke et al., 2017). 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. 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 6 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 6). This can be done efficiently, e.g. using a lexicographic sort of the packed bitmask representing the sets of visited nodes[7].

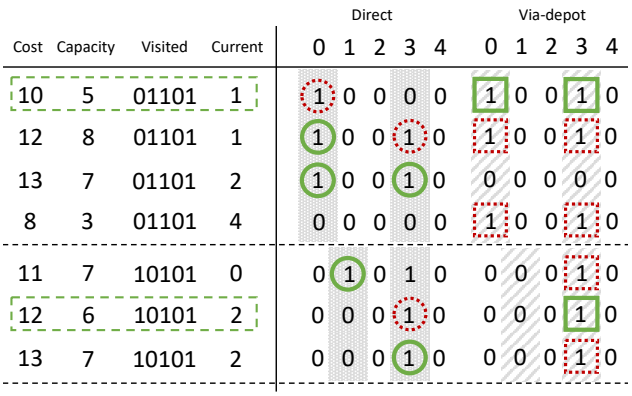

Figure 6. Example beam for VRP with variables, grouped by set of visited nodes (left) and feasible, non-dominated expansions (right), with $2n$ columns corresponding to n direct expansions and n via-depot expansions. Some expansions to unvisited nodes are infeasible, e.g. due to the capacity constraint 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 7 for (non-)dominated expansions corresponding to the same DP state.

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 VRP in Figure 6), 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 [8].

[7] 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.

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

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 6). 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 7).

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*[9] 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 7). 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 6). 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 7. 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 7). 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 6). For each DP state, the lowest cost entry is never dominated. The other entries 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 7).

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 this 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.[10]

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

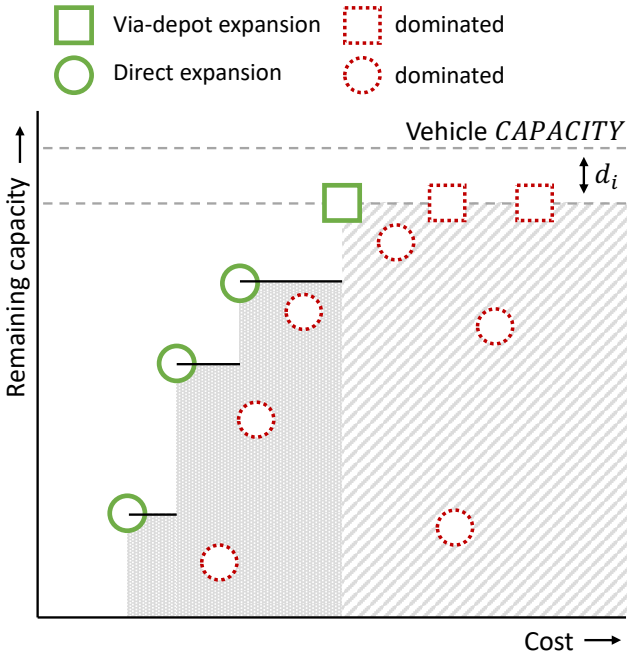

Figure 7. Example of a set of dominated and non-dominated expansions (direct and via-depot) corresponding to the same DP state (set of visited nodes and expanded node i) for VRP. Non-dominated expansions have lower cost or higher remaining capacity compared to all other expansions. The right striped 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.

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.

[10] 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.