SOLO: Search Online, Learn Offline for Combinatorial Optimization Problems

Joel Oren[1], **Chana Ross**[1], **Maksym Lefarov**[1], **Felix Richter**[1], **Ayal Taitler**[2], **Zohar Feldman**[1], **Dotan Di Castro**[1], **Christian Daniel**[1]

[1] Bosch Center for AI
[2] Technion – Israel Institute of Technology

{joel.oren, chana.ross}@il.bosch.com, {maksym.lefarov, felixmilo.richter}@de.bosch.com
ataitler@technion.ac.il, {zohar.feldman, dotan.dicastro}@il.bosch.com, christian.daniel@de.bosch.com

Abstract

We study combinatorial problems with real world applications such as machine scheduling, routing, and assignment. We propose a method that combines Reinforcement Learning (RL) and planning. This method can equally be applied to both the offline, as well as online, variants of the combinatorial problem, in which the problem components (e.g., jobs in scheduling problems) are not known in advance, but rather arrive during the decision-making process. Our solution is quite generic, scalable, and leverages distributional knowledge of the problem parameters. We frame the solution process as an MDP, and take a Deep Q-Learning approach wherein states are represented as graphs, thereby allowing our trained policies to deal with arbitrary changes in a principled manner. Though learned policies work well in expectation, small deviations can have substantial negative effects in combinatorial settings. We mitigate these drawbacks by employing our graph-convolutional policies as non-optimal heuristics in a compatible search algorithm, Monte Carlo Tree Search, to significantly improve overall performance. We demonstrate our method on two problems: Machine Scheduling and Capacitated Vehicle Routing. We show that our method outperforms custom-tailored mathematical solvers, state of the art learning-based algorithms, and common heuristics, both in computation time and performance.

Introduction

Combinatorial optimization (CO) is a central area of study in computer science with a vast body of work that has been done over the past several decades. Despite growing compute resources and efficient solvers, many practical problems are computationally intractable, and therefore, problem-specific heuristics and approximation methods have been developed (Williamson and Shmoys 2011). However, these methods suffer from two limitations. First, they are usually highly specialized: they require the algorithm designer to gain insights about the problem at hand. Second, they typically aim for worst-case scenarios; this often renders them less useful in practical applications, where average-case performance is key.

Researchers have recently studied the applicability and advantages of Reinforcement Learning (RL) methods for CO problems. In particular, and unlike analytical methods, learned policies can be evaluated in near real-time once trained, thus enabling fast response times without sacrificing the solution quality. While early work concentrated on simplified state-action spaces (e.g., when to run a given heuristic in a search procedure Khalil, Dilkina, and et al. 2017; Kruber, Lübbecke, and Parmentier 2017), recent advances in RL have enabled the training of policies on more expressive representations of the state-action space (see e.g., Waschneck, Reichstaller, and et al. 2018).

A common RL approach to CO is to incrementally construct solutions by treating the partial solutions as the *states* and their extensions as *actions*. We propose to model the state-action space as a *graph* and learn a graph neural network (GNN) policy that operates on it. GNNs have recently become a popular neural method for learning representations of rich combinatorial structures using graphs. They offer a number of desireable properties such as invariance to node permutation and independence of graph and node neighborhood sizes. Specifically in our case, they allow us to handle problems of different sizes in states and actions spaces, using the same compact network. We represent each *separate* observation as a graph, which lets us deal with dynamic changes in the problem. We then extend existing Deep Q-learning approaches (DQN; Mnih, Kavukcuoglu, and et al. 2013; Hessel, Modayil, and et al. 2018) to learn effective policies that are agnostic to the number of components in the problem instance (e.g., locations in routing problems, jobs in scheduling problems), and compatible with online stochastic arrivals. We train our GNN models to predict the Q-values of states, by making use of their applicability to graphs of varying sizes. We note at this point that exploring GNN architectures was not the main focus of this work, but utilizing a common architecture and in our planning-learning scheme.

Combinatorial problems are notorious for being sensitive to slight perturbations in their solutions (see e.g., Hall and Posner 2004 and Chapter 26 in Gonzalez 2007). Drawing inspiration from recent self-play methods for board games (Silver, Schrittwieser, and et al. 2017), we complement our policy with a search procedure, using our trained Q-network as a guide in a modified Monte-Carlo Tree Search (MCTS). We demonstrate the efficacy of our hybrid method, on two NP-hard problems (Kramer, Iori, and Lacomme 2021; Toth and Vigo 2014) and their *online* variants: the Parallel Ma-

chine Job Scheduling problem (PMSP), and the Capacitated Vehicle Routing problem (CVRP).

Contributions: We propose a method that incorporates end-to-end GNN models. This allows our trained, fixed-size policies to operate on problems with varying state and action sizes. In particular, our approach is capable of solving *offline* and *online* problems of magnitudes that are not known a priori. To mitigate degradation in performance of learned solutions, we devise a scheme that bridges the gap between searching online and learning offline, named *SOLO*. This approach benefits from the generalization capacity, and real-time performance of trained policies, as well as the robustness obtained by online search. Our results show that SOLO finds higher quality solutions than existing learning-based approaches and heuristics, while being competitive with fine-tuned analytical solvers.

Related Work

The application of Machine Learning (ML) approaches to NP-hard problems (Garey and Johnson 1990) has gained much interest recently. While they are generally hard to solve at scale, these problems are amenable to approximate solutions via ML algorithms (Bengio, Lodi, and Prouvost 2021). In general, there are three popular approaches for solving NP-Hard combinatorial problems. The most basic one includes non ML algorithms such as branch & bound search. Another, applies ML techniques in an end to end manner. Finally, there are works that combine the two previous methods together in an augmented way.

Classical Search Methods: Researchers in Operations Research and Computer Science have developed an extensive set of tools and algorithms for CO problems over the years (Korte et al. 2011). Their drawbacks are usually the need for domain-specific tailoring or extensive computational requirements (Wolsey and Nemhauser 1999). Peters, Stephan, and et al 2019 compared local search via genetic algorithms and Integer Programming in a staff assignment problem, and showed that local search can produce sub-optimal solutions relatively fast. Many other local search methods using neighborhood relations have been established, and applied to problems such as paint shop scheduling (Winter, Musliu, and et al. 2019), scheduling with time widows (He, de Weerdt, and Yorke-Smith 2019) and more. A popular approach for global search is Monte-Carlo Tree Search (MCTS). The Upper Confidence-Bounds applied to Trees (UCT) (Kocsis and Szepesvári 2006) is a particular instance of MCTS, which provides formal guarantees on optimality and achieves attractive results in practice. Distributed approaches have also been investigated by the community, Nicolo, Ferrer, and et al. 2019 solved a multi-agent scheduling problem using a distributed structure. They have allocated different agents that execute simultaneous local searches on previously found solutions.

End To End Machine Learning: Common approaches apply RL algorithms in order to generate solutions without the use of problem specific heuristics. Khalil, Dai, and et al. 2017 implemented a combined DQN and GNN architecture in order to learn a greedy search for graph optimization problems. Bello et al. 2016 used an RNN and REINFORCE (Williams 1992) to train an agent to solve the Travelling Salesperson problem (TSP). Nazari, Oroojlooy, and et. al. 2018 employed a similar method with the addition of an attention mechanism. Kool, van Hoof, and Welling 2019 also approached routing problems using a GNN architecture combined with an attention encoder-decoder mechanism and the REINFORCE algorithm. These three works employ a *masking* step on their outputs to disallow unavailable actions (e.g., locations already visited). This approach requires the set of actions to be set in advance. Mao, Schwarzkopf, and et al. 2019 also used REINFORCE with graph convolutional embedding for job scheduling on a cluster, but in an online setting where the jobs had a DAG structure that could be exploited. No search was incorporated in here due to the real-time dynamic nature of the processing cluster scheduling. Recently Joe and Lau 2020 applied DQN in the Dynamic Vehicle Routing Problem to learn approximated value function and a routing heuristic.

Combined Search and Machine Learning: Combination of search algorithms and ML techniques can benefit one another by either employing search to accelerate the learning, or learning better models for the search to use or both. Waschneck, Reichstaller, and et al. 2018 applied RL methods for optimizing scheduling problems with a multi-agent cooperative approach. However, their experiments did not demonstrate clear improvements over heuristic algorithms. Chen and Tian 2019 took a different approach, they used a DQN for choosing regions in the solution to improve a local search heuristic. We use their method as a baseline in our experiments section. Zhuwen, Qifeng, and Vladlen 2018 used GNN and supervised learning to label nodes in a graph, to determine whether they belong to a Maximal Independent Set. This prediction was used within a tree search algorithm to find the best feasible solution the network predicted. Recently, MCTS combined with RL methods has gained much traction due to superhuman play level in board games such as Go, Chess, and Shogi (Silver, Schrittwieser, and et al. 2017). Laterre, Fu, and et al. 2018 have integrated MCTS into a RL loop and solved the Bin Packing Problem (BPP) in two and three dimensions using ranking rewards.

Method and Definitions

Modeling Combinatorial Optimization Problems

A combinatorial optimization (CO) problem is given by a triplet $\langle \mathcal{I}, S, f \rangle$, where $\mathcal{I}$ is the set of problem instances, S maps an instance $I \in \mathcal{I}$ to its set of feasible solutions, and f is the objective function mapping solutions in $S(I)$ to real values. We model a sequential solution process of an instance I, in which at each time t a partial solution is extended, using a finite horizon Markov Decision Process (MDP; Puterman 1994) of T steps. At each time t, the state s_t corresponds to a partial solution, an action a_t corresponds to a feasible extension of s_t (similarly to a greedy algorithm), a reward $r_{t+1} = r(s_t, a_t) = f(s_{t+1}) - f(s_t)$,[1]

[1]Note that a different reward function that equals the resulting solution value at the end of the episode, and zero anywhere else

and a transition probability $p(s'|s_t, a)$. The action distribution is set by a policy $\pi(a|s)$. This leads to a distribution over *trajectories* $\rho = (\langle s_t, a_t, r_{t+1}\rangle)_{t=0,\ldots,T-1}$, $p(\rho) = p(s_0)\prod_{t=0}^{T-1}\pi(a_t|s_t)p(s_{t+1}|s_t, a_t)$. The Q-function is defined as

$$Q(s_t, a_t) \triangleq \mathbb{E}_\rho\left[\sum_{i=0}^{N-t} r(s_i, a_i)\middle| s_0 = s_t, a_0 = a_t\right],$$

where the agent's objective is to find an optimal policy $\pi^*(a|s) = \arg\max_a Q(s, a)$ (Sutton and Barto 2018).

We encode a problem instance I, or rather, a sub-problem thereof, induced by a partial solution s_t, as a graph $G_t = (V_t, E_t, f_t^v, f_t^e)$ where f_t^v (resp. f_t^e) maps nodes (edges) to feature vectors. This encoding is often quite expressive and allows for a systematic treatment of many CO problems.

Motivating Combinatorial Problems

Our approach is quite generic and can be applied to many optimization problems. We illustrate its efficacy on two well-known problems: the Parallel Machine Scheduling Problem (Kramer, Iori, and Lacomme 2021), or PMSP for short, and a simplified version of the Capacitated Vehicle Routing problem (CVRP), introduced by Dantzig and Ramser 1959. Both PMSP and the CVRP are widely-known problems, however, we briefly outline them here.

CVRP is widely known, but briefly: there is a set $N = \{1, \ldots, n\}$ of customers with demands $\{d_i\}_{i\in N}$ for a single commodity, and locations $\{p_i\}_{i\in N}$. The commodity is located at a depot o, and is to be transported from it to the customers so as to meet their demand by a vehicle of capacity C^* (for feasibility, we assume that $d_i \leq C^*$ for all $i \in N$) and velocity ν. A *tour* consists of visited locations, $\tau = (p_1 = o, \ldots, p_{n_\tau} = o)$, in which the vehicle's load does not exceed its capacity and satisfies the demands of the customers along it. The objective is to minimize the total distance traveled by the vehicle.

PMSP is a scheduling problem with m (unrelated) machines, and n jobs with weights $\left(w_i\right)_{i=1,\ldots,n}$ and processing times $\left(p_i\right)_{i=1,\ldots,n}$. Also, each job u belongs to a job class $\kappa_u \in \{1, \ldots, c\}$, so that if job u is scheduled to process immediately after job u' on the *same* machine, then there is an additional incurred *setup time* $P[\kappa_{u'}, \kappa_u]$, where P is a matrix with non-negative entries, and zeros in the diagonal. The objective is to minimize the total *weighted* completion time, which consists of the waiting, setup, and processing times of all arrived jobs.

Online Variants: Both problems have online variations, in which the problem constituents are *not* known apriori, but rather, arrive according to certain distributions, at the time in which a solution is constructed. In the case of online CVRP, customers arrive at different times, as the vehicle proceeds along its current route. In PMSP, jobs appear as machines are processing the previously arrived jobs.

Solution Approach

Our solution is composed of two phases as depicted in Figure 1. First, we train an agent offline, using a simulated environment, in an off-policy manner, where representations of the states are learned using a graph neural network. Second, at solution time, we use the policy trained offline as a heuristic in a Monte-Carlo tree search (MCTS). This approach allows us to train offline a good policy, while the online tree search ensures a more robust solution.

Learn Offline: We base our offline part of the method on the off-policy DQN algorithm where we learn a neural approximation $\tilde{Q}(s, a; \theta) \approx Q(s, a)$, parametrized by θ (Mnih, Kavukcuoglu, and et al. 2013). Modelling $\tilde{Q}$ using feed-forward or convolutional neural network architectures is challenging in the CO setting as the sizes of the state and action spaces can vary throughout an episode. For this reason, we encode a state-action tuple as a graph $G_t = (V_t, E_t, f_t^v, f_t^e)$ with sets of nodes V_t, edges E_t, node features f_t^v and edge features f_t^e. See the next section for problem-specific descriptions of the graph encodings.

As shown in Figure 1, we train our $\tilde{Q}$ model using a simulation of the problem. At decision time t, the current state s_t is translated to a graph, and the next action is chosen using the model's prediction of the Q-values for each state-action tuple (s_t, a_t). We take an epsilon-greedy approach in the training phase for choosing the next action. We model $\tilde{Q}$ as a Graph Neural Network (GNN) with the following three components: (1) embedding model: a vanilla feed forward model with leaky *ReLU* activation function. This component converts the features of each node to a higher dimension. (2) encoder model. (3) decoder model. The encoder and decoder models both use a variant of the message-passing GNN architecture (Battaglia, Hamrick, and et al. 2018) with the following update rules:

$$\begin{aligned}
e_i^{(t+1)} &= \phi_e(e_i^{(t)}, \rho^{v\to e}(V), w), \forall e_i \in E,\\
v_i^{(t+1)} &= \phi_v(v_i^{(t)}, \rho^{(e,v)\to v}(V, E), w), \forall v_i \in V,\\
w^{(t+1)} &= \phi_w(w^{(t)}, \rho^{(e,v)\to w}(V, E)),
\end{aligned}$$

Where $\rho^{v\to e}$, $\rho^{(e,v)\to v}$, $\rho^{(e,v)\to w}$ are the aggregation functions that collect information from the *related* parts of G_t, and ϕ_e, ϕ_v, ϕ_w are the neural networks that update the feature vectors f_t^v, f_t^e and w, respectively (w being a global feature of the graph).

Each neural network consists of a number of fully connected layers (three layers in the encoder model and one in the decoder model), followed by a leaky *ReLU* activation function and finally a normalization layer. Our DQN training process includes several common techniques used in deep Q-learning, such as double DQN and priority replay (see e.g., Hessel, Modayil, and et al. 2018). More information about our model can be found in the supplementary material.

We compute $\tilde{Q}(s, a)$ as the the output of ϕ_e at the final message-passing step of the decoder model since the edges

would tend to make the learning process harder, due to its sparsity.

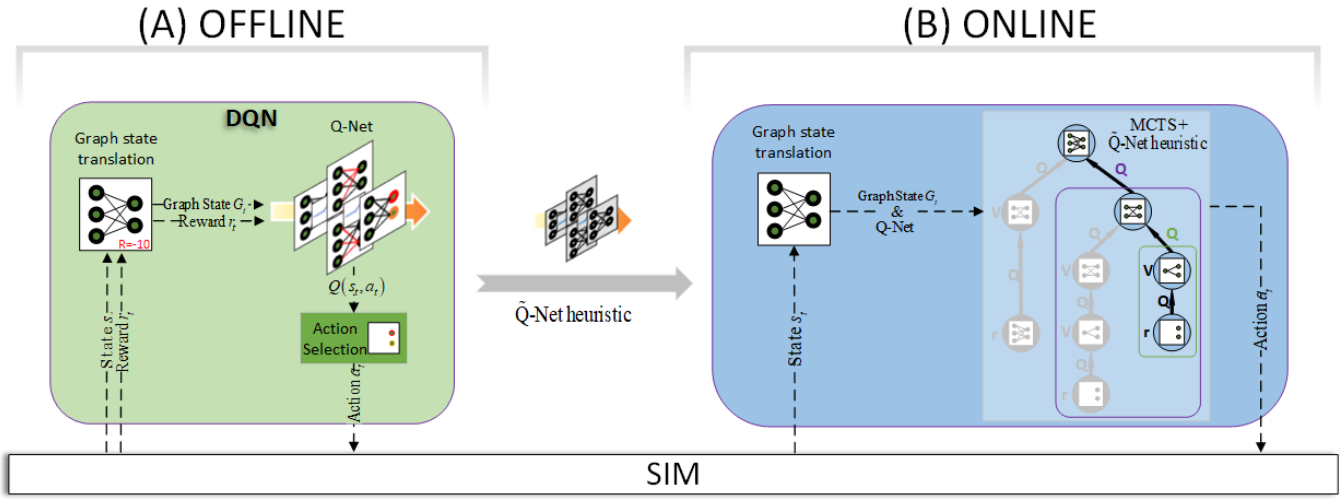

Figure 1: A schematic overview of SOLO. On the left, a depiction of our DQN training process, which produces the $\tilde{Q}$-Net heuristic. On the right is our planning procedure that, for each step, runs our modified MCTS with $\tilde{Q}$-Net as a heuristic.

represent the actions in our problems. This architecture allows us to address the changing size of the state (i.e., different $|V|$ and $|E|$ per step) in a principled manner. A nice property of GNN's is that the results are insensitive to ordering in the input, making them particularly effective for un-ordered data such as sets and graphs.

Search Online: At run time, we use online search to optimize the decisions in the states observed in real time. Namely, we take the time until the next decision is due (e.g., the next job assignment or the next customer pick-up) to evaluate all the applicable actions in the current state, and upon reaching the timeout, output the estimated best action for this state. We employ a modified version of the popular MCTS algorithm UCT (Upper Confidence-Bound applied to Trees, Kocsis and Szepesvári 2006) in conjunction with the $\tilde{Q}$-value that was trained offline. The algorithm works by sampling rollouts iteratively from the current root state, s_t, until we reach a terminal state. After each iteration of the online search, the Q-value of any previously encountered state-action pair (s, a) is estimated as $Q^{\mathcal{T}}(s,a) = \sum_{i=1}^{n(s,a)} \frac{\overline{r}_i}{n(s,a)}$, with $n(s, a)$ denoting the number of times action a was sampled in state s over all the sampled rollouts. Each reward-to-go $\overline{r}_i$ is the accumulated reward from state s onward in a corresponding rollout that passed through (s, a). Similarly to UCT, we select actions during each rollout according to the Upper Confidence-Bound (UCB) policy. Namely, we select an action that was not yet sampled in the current state, if such action exists, and otherwise we select the action that maximizes the value $Q^{\mathcal{T}}(s,a) + \beta\sqrt{\frac{\log(n(s))}{n(s,a)}}$, with $n(s) = \sum_a n(s,a)$, and β is the exploration factor that governs the balance between exploration and exploitation. Algorithm 1 depicts our online search. While UCT is guaranteed to asymptotically converge at the optimal decision by sampling each action infinitely often, in order to improve its practical efficiency, we combine it with the following features:

MCTS heuristic: We use the $\tilde{Q}$-value as an out-of-tree heuristic. We employ an iterative deepening scheme according to which, in each iteration, the search tree is expanded with at most one state, which is the first state in the rollout that is not already in the tree. During a rollout, whenever we encounter states that are not yet in the tree, we choose the action that maximizes the $\tilde{Q}$-value rather than using UCB (corresponding to a random action in this case). This is a common technique that often improves performance significantly, depending on the quality of the used heuristic.

Action pruning: Second, we use the $\tilde{Q}$-values to *prune the action space*. Since the number of plausible actions in CO problems tends to be intractably high, we would like to focus the samples on actions that have higher potential to be a (near-)optimal action. We thus consider for each state s only the k actions with the highest $\tilde{Q}$-value, where k is a hyper-parameter of our algorithm. Indeed, as the estimated $\tilde{Q}$-value becomes more accurate, the probability that one of these k actions is near-optimal increases.

Rollout preemption Additionally, to allow for more sampled rollouts, we shorten the lookahead period by suppressing future arrivals (e.g., arrivals of jobs or customers) that are scheduled for a time after $t + \Delta_T$, where Δ_T is a hyper-parameter). As this technique is mostly implemented in the simulation, it is not depicted in Algorithm 1. By applying the above features, we theoretically compromise optimality, but our experiments demonstrate that they hold some practical merits.

Implementation Details

In the following section we show how to represent states, actions, and rewards in our solution framework.

Graph States: In *CVRP* a state s_t is given by the set of n_t

pending customers, their coordinates $\mathbf{p_i} \in \mathbb{R}^2$ and demands $d_i \geq 0$, for $i = 1, \ldots, n_t$. The current state also specifies the current vehicle location $\mathbf{p}^*$, along with its remaining capacity c^*, and the location of the depot $\mathbf{p_o}$. To encode s_t, we use a simple star graph topology with $n_t + 2$ nodes that consist of a vehicle node u^* as the central node that is connected to n_t customer nodes $\{u_i\}_{i=1,\ldots,n_t}$ and a depot node u_o. We employ a unified node feature vector that holds entries for the features of all three node types (with zeros where inapplicable): node u's location p_u, capacity c^* (vehicle), and demand (customers). A node feature vector also includes a length-3 one-hot encoding denoting the node type. Finally, a node u_i's feature vector contains a binary value $\mathbb{I}[d_i \leq c^*]$, indicating (in the case of a customer) whether the vehicle can travel to it. As for edges connecting the customer to the depot and vehicle, their features consist of solely their respective distances.

In *PMSP*, a state s_t consists of the currently pending n_t jobs, the set of m machines along with the remaining processing times $(r_t^{(i)})_{i=1,\ldots,m}$ and their last or currently assigned job classes (for computing setup times), $(\kappa_{t,i}^{last})_{i=1,\ldots,m}$. We represent this as a complete bipartite graph $(V_t = (J_t, M), E_t, f_t^v, f_t^e)$, where J_t and M are the sets of job and machine nodes, respectively, $E_t = J_t \times M$ is the complete set of edges, connecting every job to every machine. and lastly, f_t^v and f_t^e map nodes and edges to feature vectors. For a job node $u_j \in V_t, j = 1, \ldots, n_t$, its feature vector $f_t^v(u_j)$ consists of its processing time p_j, weight w_j, and arrival time a_j (positive in the online case), a length-c one-hot encoding indicating the job class κ_j, and a node-job indicator $\mathbb{I}_{u_j}^{job} = 1$. For a machine node $v_i \in M, i = 1, \ldots, m$, in addition to a machine indicator bit $\mathbb{I}_{v_i}^{job} = 0$, we also encode the remaining processing time of the machine $r_t^{(i)}$, and a one-hot encoding of length $c + 1$ specifying the class of last run job, $\kappa_{t,i}^{last}$ (including the special "empty class" for machines that were not yet assigned jobs). For an edge e_{ji}, connecting job node u_j to machine node v_i, the only feature is the incurred setup time $P[\kappa_j, \kappa_{t,i}^{last}]$.[2] As before, we keep the node feature vectors at equal lengths, by having every node vector contain entries for both job and machine features; for job nodes the machine feature entries are zeroed out, and vice versa. Figures 2 and 3 depict the resulting complete bipartite graph and feature vectors, respectively.

Actions: In both settings, the set of actions are specified by the set of edges in the graph representation of the current state s_t: in CVRP, selecting an edge between u^* and u_i (resp. u_o), corresponds to extending the route traveled thus far by instructing the vehicle to drive to a customer (resp. depot) in location $\mathbf{p}_i$ (resp. $\mathbf{p_o}$). Note that an edge to a customer node u_i is *feasible* provided that $d_i \leq c^*$. Similarly in PMSP, the set of actions is specified by the edges connecting jobs to machines. Selecting an edge (u_j, v_i) corresponds to assigning job j to machine i at the time of state s_t. Importantly, in both settings we also allow for the special *noop* action that simply has no effect apart from "skipping" to the next decision point (see below). Additionally, a given state might induce a graph representation with certain disallowed edge actions. Though one can simply remove these edges from the graph, we decided to leave them in, and apply action masking. We found this approach to be more effective since it enables better flow of information between the nodes, and gave overall better results.

Decision Events: In CVRP a decision point at time t occurs when the vehicle has reached its previous destination, or in the online setting where a *noop* action was previously

[2] If machine i was not previously assigned a job, than the setup time would be zero.

Algorithm 1: MCTS with pruned action search

Input: State S^t, Environment env, Q-value estimator $\tilde{Q}$, random seed $seed$, rollout budget r_max, time budget t_max, prune limit k, state visits counter n_s, state-action visits counter n_{sa}, Search Tree estimator $Q^{\mathcal{T}}$

env.set_seed(seed)
for $r \leftarrow 1, \ldots, r\_max$ **do**
  if *elapsed_time()* $> t\_max$ **then**
    break
  $s \leftarrow$ env.set_state(S^t)
  $done, \rho \leftarrow false, []$
  $t_{out} \leftarrow \infty$
  $t \leftarrow 0$
  while *not done* **do**
    $t \leftarrow t + 1$
    $\tilde{q} \leftarrow \tilde{Q}(s, \cdot)$
    if $n_s(s) == 0$ **then**
      `// action with max. Q-value`
      $a \leftarrow \arg_a \max(\tilde{q}[a])$
      $t_{out} \leftarrow \min(t, t_{out})$
    else
      $top\_actions \leftarrow \text{top\_k\_args}_a(\tilde{q}[a], k)$
      $q_{UCB} \leftarrow \{\}$
      for $a \in top\_actions$ **do**
        $q_{UCB}[a] \leftarrow Q^{\mathcal{T}}(s,a) + \beta\sqrt{\frac{\log n_s(s)}{n_{sa}(s,a)}}$
      $a \leftarrow \arg_a \max(q_{UCB}[a])$
    $r, s', done \leftarrow$ env.step(a)
    $\rho[t] \leftarrow \langle s, a, r \rangle$
    $s \leftarrow s'$
  update_tree$(\rho, n_s, n_{sa}, Q^{\mathcal{T}}, t_{out})$
return $\arg_a \max\left(Q^{\mathcal{T}}(S^t, a)\right)$
Def `update_tree`$(\rho, n_s, n_{sa}, Q^{\mathcal{T}}, t_{out})$**:**
  $\overline{r} \leftarrow 0$
  for $i \leftarrow |\rho|$ *to* 1 **do**
    $s, a, r \leftarrow \rho[i]$
    $\overline{r} \leftarrow r + \gamma\overline{r}$
    if $i > t_{out}$ **then**
      continue
    $n_s(s) \leftarrow n_s(s) + 1$
    $n_{sa}(s,a) \leftarrow n_{sa}(s,a) + 1$
    $Q^{\mathcal{T}}(s,a) \leftarrow \frac{n_{sa}(s,a)-1}{n_{sa}(s,a)} Q^{\mathcal{T}}(s,a) + \frac{1}{n_{sa}(s,a)}\overline{r}$

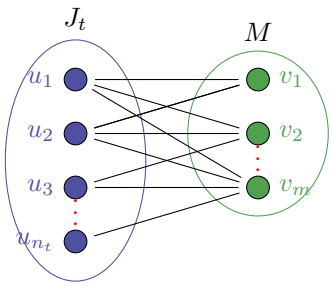

Figure 2: PMS: The GNN representation of a state S_t with n_t waiting jobs and m machines.

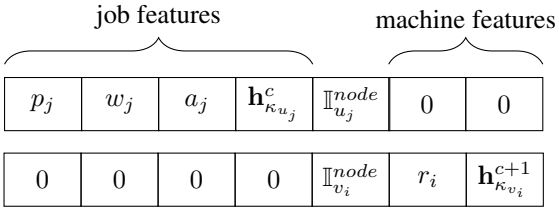

Figure 3: Feature vectors for job node u_j and machine node v_i (PMSP).

taken and a new customer arrives. In PMSP, a decision point at time t occurs when a machine has become free and there is at least one pending job, or conversely: a new job has arrived since the previous decision point (in the online setting), and there is a free machine.

Reward Function: Both CVRP and PMSP are minimization problems, and hence each time t would incur a cost, or a negative reward, taken to be the difference in the objective value for the partial solution resulting from extending the current partial solution by taking a_t. In CVRP, this translates to simply the distance traveled between step t and $t+1$. In PMSP, it is the product of the total weight of the jobs processed between steps t and $t+1$ and their time difference.

Empirical Evaluation

We evaluate SOLO on the online and offline variants of the two problems mentioned. With minor changes in the graph representation and small hyper-parameter tuning, we manage to achieve competitive results in the online setting without degrading the offline results.

Experimental Setup

As mentioned in previous sections all problems are represented as a bipartite graph (in CVRP we have a simplified star graph) where the actions are edges and the agents or assignments are nodes. We compare all problems to known baselines including naive solutions, optimal solutions (CPLEX or OR-Tools), problem tailored heuristics and a learned RL solution (Neural Rewriter, Chen and Tian). More information about the baselines can be found in the supplementary material. For fair comparison, we use the same hardware and training time for all networks trained (our Q-net and Neural Rewriter).[3][4] In addition, MCTS and optimization algorithms are given 10 seconds to optimize each decision.

The known algorithms for online problems are typically designed to minimize the competitive ratio, i.e., the worst-case ratio of the an algorithm's performance to that of the optimal solution in hindsight (see e.g., Jaillet and Wagner for a survey). However, the worst-case theoretical nature of the measure makes it less suitable in more practical scenarios. Therefore, the online results are compared to the offline baselines while running them in a quasi-offline manner: at each planning step we re-run the offline algorithm including only opened jobs or customers. In CVRP this translates to solving an offline problem whenever the vehicle reaches the depot (all offline baselines assume the vehicle starts the route and the depot and therefore could not be used each time we reach a customer). In PMSP, we solve a new offline problem after each interval or when a machine becomes available.

To better understand the contribution of our approach and components of our solution, we evaluate SOLO in three ways:

1. Trained $\tilde{Q}$ alone (Offline part of SOLO)
2. MCTS + Naive baseline as heuristic (Online part of SOLO)
3. MCTS + Trained $\tilde{Q}$ as heuristic (Full SOLO solution)

For the online problems, we run MCTS with rollout preemption, where in each rollout, the sampling of customers or jobs is stopped past a certain time. In addition, we use an action pruning mechanism to only evaluate a subset of the possible actions at each step. Given the limited computing time, these techniques allow for a good balance between planning for the future on the one hand, but also seeing a broad view of the current state and possible actions. We train our policy, Q-net, which is based on $\tilde{Q}$, using a decaying learning rate with an initial value of 10^{-3}. To allow for exploration the learning starts after $5{,}000$ steps and an ε function is used, where the initial value of ε is 1.0 and it decays linearly over time. The training stops after 10^6 steps and the model that achieves the best overall evaluation is saved and used. see the supplementary material for the full list of hyper-parameters used.

CVRP: In both online and offline cases, we consider the same setting as previous work (Chen and Tian; Kool, van Hoof, and Welling; Nazari, Oroojlooy, and et. al.) that consider three scenarios with $N \in \{20, 50, 100\}$ customers and vehicle capacities $C \in \{30, 40, 50\}$, respectively. Locations are sampled from a uniform distribution over the unit interval, and demands from a discrete uniform distribution over $\{0, \ldots, 10\}$.

[3] 18-core Intel Xeon Gold 6150 @ 2.70 GHz, Nvidia Tesla V100 (16 GB)

[4] all policies were trained for less than 24 hours

In the online problem we introduce an arrival time in addition to the position and demand of each customer. Unlike the offline case, here the customer parameters are sampled from a Truncated Gaussian Mixture Model (TGMM). This sampling shows the strength of RL algorithms like ours, which take into account future customers by learning the distribution during training.

We compare our offline results to the following baselines: a trained Neural Rewriter model (Chen and Tian), the two strongest problem heuristic baselines reported by Nazari, Oroojlooy, and et. al. (Savings and Sweep), Google's OR-Tools, and the naive Distance Proportional baseline. In the latter baseline, each valid customer (whose demand is lower than the current capacity) is selected with probability proportional to its distance from the vehicle.

PMSP: For the offline problem we consider several cases with 80 jobs, 3 machines, and 5 job classes. We train a model on problems where job parameters are sampled from discrete uniform distributions (more details can be found in the supplementary material). In the online setting we run two problems, both with 80 jobs in expectation and 5 job classes. The first problem consists of 3 machines and jobs arrive in 16 intervals of length 130 time units each. The second problem consists of 10 machines and 60 intervals with 10 time units. We sample job processing times, setup times and job weights from discrete uniform distributions. Unlike the offline setting, here jobs arrive according to Poisson distributions, where the frequency of the i-th class is proportional to $1/i$. The problem parameters were chosen so as to not overload the machines in steady state, while maintaining relatively low levels of idleness.

We evaluate the offline problem on a public set of benchmark problems which uses the same ranges for job features (Liao, Chao, and Chen 2012). We compare to the following baselines: a heuristic that is a variant of the weighted shortest processing times first heuristic (WSPT), the Neural Rewriter approach by Chen and Tian, and the solution found after a fixed time using the IBM ILOG CPLEX CPOptimizer (V12.9.0) constraint programming suite. More details about the baselines can be found in the supplementary material. In addition to the public benchmark we evaluate our solution on a smaller problem with 20 jobs, where the optimal solution is known.

Experimental Results

For each problem we train a Q-net model and run a comparison on 100 seeds between our method and the baselines mentioned above.

CVRP: Both online and offline results can be found in Table 1, more experiments are detailed in the appendix (Oren, Ross, and et al. 2021). In the online problem SOLO outperforms all the other algorithms. In addition, despite SOLO being an algorithm for online problems, it reaches decent performance for the offline problem. When comparing SOLO to NeuralRewriter, the other learning algorithm, SOLO performs better and shows improved results.

We notice that the Savings algorithm achieves slightly better results than SOLO, which is expected since it is a problem specific heuristic and tailored to this problem. However, unlike SOLO, Savings does not extend to the online problem well and performs weakly when customers arrive over time.

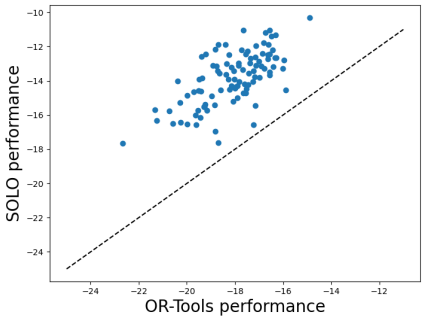

Figure 4: A performance comparison of SOLO and OR-Tools in the online CVRP50 settings.

When examining all the variants of our method we notice a number of interesting results. First, Q-net alone achieves good results compared to other algorithms, showing that the RL algorithm with the GNN architecture provides a good solution policy. Second, MCTS alone shows good results even when combined with a random heuristic, showing the value in online search algorithms and rollouts. Last, combining MCTS with our Q-net model achieves the best results showing the full strength of our method and necessity for both the offline learning and online search. Note that in all CVRP cases, SOLO with pruning did not show any improvement on SOLO alone and we decided not to include this in the plots.

In Figure 4 we see a comparison between SOLO and OR-Tools for the online problem with 50 customers. Each point above the dashed line is an instance where SOLO outpreformed OR-Tools. We see in the figure that all instances run in this evaluation are above the line and have better results when using SOLO.

PMSP: We evaluate the model trained on 20 jobs on a public set of offline benchmarks where the optimal solution quality is known (Kramer, Iori, and Lacomme 2021). The results in Table 2 show our approach performs mostly on par with CPLEX. In addition, our method out-performs the other learning algorithm (Neural Rewriter) and shows good results when evaluating on 20 and 80 jobs. Similar to CVRP, MCTS improves the results achieved by Q-net alone, and shows that adding an online search algorithm to the offline learned model improves results. The scatter plot in Figure 5 shows that in the online scheduling problem, our approach yields better solutions than WSPT in all problem instances.

To analyze whether the Q-net model actually learns to take advantage of the arrival distribution, we apply it in a scheduling setting where we flip the class identity of observed jobs from i to $5 - i$. The results shown in Figure 6 indicate that the model is indeed optimized for the arrival distribution used during training.

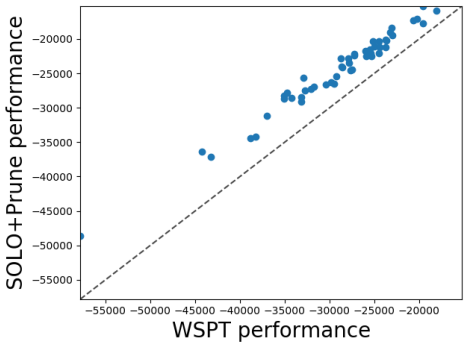

Figure 5: A performance comparison of SOLO+Prune and WSPT in the online setting with 10 machines and 50 random instances.

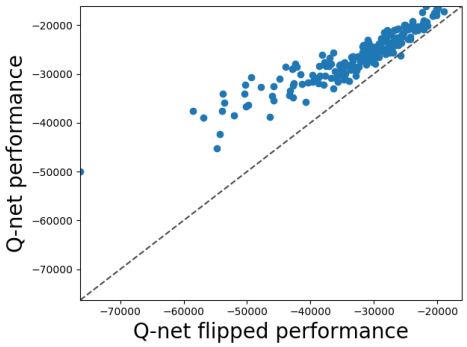

Figure 6: Performance comparison of Q-net to its performance in the online setting with "flipped classes". 10 machines, 60 arrival intervals, interval durations of 10, and 200 random instances.

Conclusion and Discussion

We presented a deep RL approach to combinatorial optimization. Our method incorporates graph-based representation of the state-action space into Deep Q-learning to learn an input-size agnostic policy, and we further combine it with MCTS to significantly improve performance of our approach. We evaluated these contributions on two combinatorial optimization problems, PMSP and CVRP. Our pure Q-net agent, which provides near-instantanous action selection, outperforms popular heuristics. Conversely, our combined approach, SOLO, makes full use of any available time for deliberation thanks to its anytime nature, effectively decreasing the gap to dedicated combinatorial optimization solvers. As a future work direction we propose to investigate explicitly learning the arrival distribution of the online combinatorial optimization problem during training. Another direction is to explore alternative graph representation for the states.

Offline CVRP		
	20	**100**
Uniform-Random[UR]	-13.21(107.51%)	-58.84 (230.13%)
Distance[D]	-10.43 (63.65%)	-47.59 (167.38%)
Savings	-6.35 (-1.04%)	**-16.51 (-7.94%)**
Sweep	-8.89 (39.33%)	-28.24 (58.11%)
OR-Tools	-6.42 (0.00%)	-17.96 (0.00%)
NeuralRewriter	-6.95 (8.48%)	-19.45 (8.57%)
Q-Net	-6.84 (6.59%)	-19.27 (7.62%)
MCTS+UR	-7.65 (19.45%)	-46.34 (160.11%)
MCTS+D	-7.15 (12.01%)	-44.00 (147.44%)
SOLO	**-6.21 (-3.18%)**	-17.68 (-1.24%)
Online CVRP		
	20	**100**
Uniform-Random[UR]	-12.72 (31.67%)	-52.73 (108.06%)
Distance[D]	-9.75 (0.76%)	-33.65 (32.72%)
Savings	-9.90 (0.51%)	-25.15 (-0.90%)
Sweep	-11.16 (13.73%)	-29.52 (16.16%)
OR-Tools	-9.86 (0.00%)	-25.40 (0.00%)
NeuralRewriter	-10.00 (1.56%)	-25.85 (1.90%)
Q-net	-8.79 (-9.76%)	-26.80 (5.70%)
MCTS+UR	-7.80 (-20.27%)	-28.72 (12.98%)
MCTS+D	-6.78 (-30.84%)	-25.58 (0.78%)
SOLO	**-6.63 (-32.38%)**	**-24.80 (-2.28%)**

Table 1: Offline and Online CVRP results. Each cell contains the average cost and the fractional improvements over OR-Tools (negative numbers are better than OR-Tools). Best results and our methods are marked in bold

Offline PMSP		
	liao 20	**liao 80**
WSPT	-16570.16 (5.82%)	-182357.02 (4.15%)
CPLEX	-15658.46 (0%)	-175084.88 (0%)
NeuralRewriter	-16540.28 (5.63%)	-182450.02 (4.21%)
Q-net	-15906.32 (1.58%)	-178444.74 (1.92%)
MCTS+WSPT	-15876.88 (1.39%)	-176439.74 (0.77%)
SOLO	-15695.94 (0.24%)	-175524.34 (0.25%)
SOLO+Prune	**-15683.46 (0.16%)**	**-175164.58 (0.05%)**
optimal	-15628.68 (-0.19%)	
Online PMSP		
	3 machines	**10 machines**
WSPT	-40601.34 (15.04%)	-29102.5 (18.87%)
CPLEX	-35294.38 (0%)	-24481.9 (0%)
NeuralRewriter	-38575.78 (9.3%)	-27350.68 (11.72%)
Q-net	-37386.9 (5.93%)	-26031.5 (6.33%)
MCTS+WSPT	-35489.56 (0.55%)	-24724.76 (0.99%)
SOLO	-35434.46 (0.4%)	-24747.38 (1.08%)
SOLO+Prune	**-35280.2 (-0.04%)**	**-24655.42 (0.71%)**

Table 2: Scheduling results for all problem variants. Each cell includes the average cost on 50 seeds and the fractional improvement of each method compared to CPLEX (negative numbers are better than CPLEX).

References

Battaglia, P. W.; Hamrick, J. B.; and et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* .

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

Bengio, Y.; Lodi, A.; and Prouvost, A. 2021. Machine learning for combinatorial optimization: A methodological tour d'horizon. *EJOR* 290(2): 405–421.

Chen, X.; and Tian, Y. 2019. Learning to Perform Local Rewriting for Combinatorial Optimization. In *NeurIPS*.

Clarke, G.; and Wright, J. W. 1964. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Oper. Res.* 12(4): 568–581.

Dantzig, G. B.; and Ramser, J. H. 1959. The Truck Dispatching Problem. *Manage. Sci.* 6(1).

Garey, M. R.; and Johnson, D. S. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co.

Garg, N.; Gupta, A.; and et al. 2008. Stochastic Analyses for Online Combinatorial Optimization Problems. In *SODA*.

Gonzalez, T. F. 2007. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman & Hall/CRC.

Hall, N.; and Posner, M. 2004. Sensitivity Analysis for Scheduling Problems. *J. Scheduling* 7: 49–83.

He, L.; de Weerdt, M.; and Yorke-Smith, N. 2019. Tabu-based large neighbourhood search for time/sequence-dependent scheduling problems with time windows. In *ICAPS*.

Hessel, M.; Modayil, J.; and et al. 2018. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI*.

Jaillet, P.; and Wagner, M. R. 2008. Online Vehicle Routing Problems: A Survey. *The Vehicle Routing Problem: Latest Advances and New Challenges* 45.

Joe, W.; and Lau, H. C. 2020. Deep Reinforcement Learning Approach to Solve Dynamic Vehicle Routing Problem with Stochastic Customers. In *ICAPS*.

Khalil, E.; Dai, H.; and et al. 2017. Learning Combinatorial Optimization Algorithms over Graphs. In *NeurIPS*.

Khalil, E. B.; Dilkina, B.; and et al. 2017. Learning to Run Heuristics in Tree Search. In *IJCAI*.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*, 282–293. Springer.

Kool, W.; van Hoof, H.; and Welling, M. 2019. Attention, Learn to Solve Routing Problems! In *ICLR*.

Korte, B. H.; Vygen, J.; Korte, B.; and Vygen, J. 2011. *Combinatorial optimization*, volume 1. Springer.

Kramer, A.; Iori, M.; and Lacomme, P. 2021. Mathematical formulations for scheduling jobs on identical parallel machines with family setup times and total weighted completion time minimization. *EJOR* 289(3): 825–840.

Kruber, M.; Lübbecke, M. E.; and Parmentier, A. 2017. Learning when to use a decomposition. In *CPAIOR*.

Kumar, M.; Dahl, G. E.; and et al. 2018. Parallel architecture and hyperparameter search via successive halving and classification. *arXiv preprint arXiv:1805.10255* .

Laterre, A. L.; Fu, Y.; and et al. 2018. Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization. In *CoRR*.

Liao, C. J.; Chao, C. W.; and Chen, L. C. 2012. An improved heuristic for parallel machine weighted flowtime scheduling with family set-up times. *Computers & Mathematics with Applications* 63(1): 110–117.

Mao, H.; Schwarzkopf, M.; and et al. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *SIGCOMM*.

Mnih, V.; Kavukcuoglu, K.; and et al. 2013. Playing atari with deep reinforcement learning. *arXiv: 1312.5602* .

Nazari, M.; Oroojlooy, A.; and et. al. 2018. Reinforcement Learning for Solving the Vehicle Routing Problem. In *NeurIPS*.

Nicolo, G.; Ferrer, S.; and et al. 2019. A multi-agent framework to solve energy-aware unrelated parallel machine scheduling problems with machine-dependent energy consumption and sequence-dependent setup time. In *ICAPS*.

Oren, J.; Ross, C.; and et al. 2021. SOLO: Search Online, Learn Offline for Combinatorial Optimization Problems. *CoRR* abs/2104.01646. URL https://arxiv.org/abs/2104.01646.

Peters, J.; Stephan, D.; and et al. 2019. Mixed Integer Programming versus Evolutionary Computation for Optimizing a Hard Real-World Staff Assignment Problem. In *ICAPS*.

Puterman, M. L. 1994. *Markov Decision Processes*. Wiley and Sons.

Rasku, J.; Kärkkäinen, T.; and Musliu, N. 2019. Meta-Survey and Implementations of Classical Capacitated Vehicle Routing Heuristics with Reproduced Results. *Toward Automatic Customization of Vehicle Routing Systems* .

Silver, D.; Schrittwieser, J.; and et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676): 354–359.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

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

Waschneck, B.; Reichstaller, A.; and et al. 2018. Optimization of Global Production Scheduling with Deep Reinforcement Learning. *Procedia CIRP* 72: 1264–1269.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *ML* 8(3-4): 229–256.

Williamson, D. P.; and Shmoys, D. B. 2011. *The Design of Approximation Algorithms*. Cambridge university press.

Winter, F.; Musliu, N.; and et al. 2019. Solution Approaches for an Automotive Paint Shop Scheduling Problem. In *ICAPS*.

Wolsey, L. A.; and Nemhauser, G. L. 1999. *Integer and combinatorial optimization*. John Wiley & Sons.

Zhuwen, L.; Qifeng, C.; and Vladlen, K. 2018. Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search. In *NeurIPS*.

Appendix

Organization The appendix is organized as follows: Section 35 presents the different baselines used for comparisons. Section 35 shows a detailed explanation of the Q-net architecture and training. Section 35 includes additional experiments and deeper explanations of results shown in Section 35. Finally, Section 35 has a theoretical example showing the added value of algorithms that learn the online distribution and use this knowledge to better solve the online problem.

In addition to this document, we supply training and evaluation source code. See the `readme.md` file in the supplementary material zip file.

Baseline Details

CVRP All our baselines share the following technique: In order to be able to apply them to online problems, we let them recompute their route every time the agent visits the depot. Conceptually, it would be possible to recompute routes at every time step, which would allow the agents to react faster to new arrivals. However, this would require substantial changes to the baseline implementations, since they all assume that the agent is at the depot when a route is computed.

We use the savings baseline implementation from the VeRyPy project (Rasku, Kärkkäinen, and Musliu 2019). Note that although this implementation differs from the original one by Nazari, Oroojlooy, and et. al. 2018 (and also used by, e.g., the Neural Rewriter paper Chen and Tian 2019), both claim to implement the same Clarke-Wright Savings algorithm. (Clarke and Wright 1964). The superior performance we report must therefore stem from differences in the implementation. For example, a performance of 12.25 is reported for the best savings variant on CVRP problems of size 50 in related work. The VeRyPy implementation achieves a value of 11.03.

For the Neural Rewriter baseline, we used the repository version. We trained models for all settings for 24 hours with the default hyperparameters. Note that the difference in performance we report on the CVRP problem compared to the original paper stems from the fact that we train Neural Rewriter models for 24h in order to try to give Neural Rewriter and our approach the same amount of training time. Training our models is always completed after at most 20 hours. For the online problems, we trained on offline problems of corresponding size where the customer positions were drawn from the Gaussian mixture distributions defined for the online settings.

The OR tools baseline is the one implemented by Nazari, Oroojlooy, and et. al. 2018. One difference is that we allow the policy to reoptimize its route every time the agent visits the depot. This is often an advantage, since it is easier to optimize a smaller remaining subproblem. The solver was executed with a 10 second time limit per optimization.

PMSP The WSPT variant we use selects, at a given state, job and machine (j, m) such that $(f(s_m, s_j) + p_j)/w_j$ is minimal, where s_m is the current class of machine m, and p_j and w_j are the processing time and weight of j, respectively.

In order to apply the Neural Rewriter approach to parallel machine scheduling we modified the authors' implementation for the scheduling problem. We represent each schedule as a single list, starting with an entry corresponding to the first machine, then entries for the jobs scheduled to the first machine, an entry for the second machine, entries for the jobs scheduled to the second machine, and so on. Initial schedules are created by applying the WSPT rule. We also tried to create initial schedules by uniformly choosing actions, with similar results. Schedules can be manipulated by swapping any two jobs. We use the same features as for our approach, with the addition of, for each job, the time it is started, the time the setup change is completed, and the time the job itself is completed. We tuned the hyperparameters with a grid search around the provided default hyperparameters. We allowed Neural Rewriter to take 50 rewriting steps and return the best schedule. We trained models for 24 hours. The training instances are sampled during training according to the ranges defined above. Since it is not possible to train the Neural Rewriter baseline directly on the online settings, we use the model trained on the 20 job offline setting instead in the online settings.

CPOptimizer was executed with a 10 second time limit on an Intel(R) Core(TM) i9-9820X CPU @ 3.30GHz with 64GB of RAM.

Problem	Network	Embedding	GNN	Output	N-passes
PMS	ϕ_e, ϕ_n	32	32 32 32	1	4
	ϕ_g	32	32 32 32	1	
CVRP	ϕ_e, ϕ_n	256	256 256 256	1	2
	ϕ_g	256	256 256 256	2	

Table 3: Output dimensions of fully connected layers and number of message passes for edge ϕ_e, node ϕ_n and global ϕ_g networks

DQN and Q-net Details

Network Architecture Details As mentioned in Section we use an embedding - encoder - decoder model where the embedding is a simple feed forward model. The encoder and decoder models are both GNN's with the same message-passing mechanism introduced by Battaglia, Hamrick, and et al. 2018. Table 3 summarizes the dimensions used in each fully connected layer and the number of passes through the GNN message passing section of the model.

Our GNN architecture does not effect the graph structure but only the nodes, edges and globali features. The final edge features are the Q-values of each action (in CVRP $e_{i,j}$ represents choosing customer j and in PSMP it represents scheduling job j on machine i). The global feature w represents the Q-value of the `noop` action[5].

DQN details Our DQN implementation includes various improvements that were suggested by Hessel, Modayil, and

[5]Note that w has dimensionality 2 for CVRP because of dueling, see Section 35

et al. 2018 such as dueling, double Q, delayed target network updates, prioritized replay and multi-step reward estimation. Once implementing all of the improvements we ran a hyper-parameter tuning algorithm (based on successive halving Kumar, Dahl, and et al. 2018) and found that some of the improvements did not demonstrate any increase in performance.

Parameter	**PMS**		**CVRP**	
	Offline	Online	Offline	Online
total time steps	$1e^6$	$1e^6$	$1e^6$	$1e^6$
target network update frequency	$5e^3$	$5e^3$	$5e^3$	$5e^3$
initial random steps	$5e^3$	$5e^3$	$5e^3$	$5e^3$
learning starts t	$5e^3$	$5e^3$	$5e^3$	$5e^3$
train frequency	1	1	1	1
discount γ	1.0	0.9	1.0	1.0
batch size	32	32	128	128
replay buffer size	$5e^3$	$5e^3$	$5e^3$	$5e^3$
prioritized replay α	0.0	0.0	$25e^{-3}$	$25e^{-3}$
prioritized replay β_0	0.4	0.4	0.4	0.4
prioritized replay e	$1e^{-6}$	$1e^{-6}$	$1e^{-6}$	$1e^{-6}$
exploration fraction	0.3	0.3	0.1	0.1
exploration final	0.1	$1e^{-4}$	$1e^{-4}$	$1e^{-4}$
learning rate	$1e^{-3}$	$1e^{-3}$	$1e^{-3}$	$1e^{-3}$
learning rate e	0.0	0.0	0.1	0.1
gradient norm clipping	$2e^2$	$2e^2$	$2e^2$	$2e^2$
double Q	`false`	`false`	`false`	`false`
dueling	`false`	`false`	`true`	`true`
n step reward estimation	1	1	1	1

Table 4: Training DQN hyper parameters used for all problems

In addition, we found that, although not warranted for theoretical reasons, introducing a discount factor helped numeric convergence in the online PMSP problem. We find that these scenarios demonstrate a high variance in the returns and make it harder for the training process to efficiently learn the Q values. The final hyper-parameters for the two benchmark problems are summarized in Table 4. Note that a different Q-net policy was trained for each problem size but all problems used the same DQN hyper-parameters.

Each training experiment was limited to 24 GPU hours. Table 5 summarizes the computational resources and training time for each problem. The time noted is the number of hours needed to achieve the best model performance (the training once models stopped improving).

The CVRP problem was able to train faster and therefore we were able to increase the batch size and enable prioritized replay buffer (this increases computational complexity of sampling and therefore can only be used when training is fast enough). In addition, we were able to increase the amount of hidden units in each fully connected layer of the GNN architecture. (specific numbers can be found in Table 4)

Evaluation Details

CVRP Online distribution The distribution chosen for the customer times in the online problem is a Truncated Gaussian Mixture distribution. Each distribution is represented by a finite number of sets: $(\mu, \sigma, \omega, a, b)$ noting the mean, standard deviation, weight, minimum value and maximum value of each truncated Gaussian distribution. In addition the position of customers is sampled from a $2d$ Truncated Gaussian Mixture distribution where (x, y) are sampled together. For details on the specific distribution values see Table 6.

Problem	**GPUh**	**Resources**
CVRP Offline size 20	13.08	18-core Intel Xeon Gold 6150 @ 2.70 GHz Nvidia Tesla V100 (16 GB)
CVRP Offline size 50	14.68	
CVRP Offline size 100	20.46	
CVRP Online size 20	5.08	20-core Intel i9-10900X @ 3.70GHz Nvidia GeForce RTX 2080 (11 GB)
CVRP Online size 50	17.05	
CVRP Online size 100	7.38	
PMS Offline size 20	14.3	18-core Intel Xeon Gold 6150 @ 2.70 GHz Nvidia Tesla V100 (16 GB)
PMS Offline size 80	21.83	
PMS Online 3 machines	6.9	
PMS Online 10 machines	10.65	

Table 5: Training time until the best model for the benchmark problems reported in GPU hours.

Distribution	μ	σ	w	a	b
position (x, y)	$(0.25, 0.25)$	$(0.1, 0.1)$	0.5	0	1
	$(0.75, 0.75)$	$(0.1, 0.1)$	0.5	0	1
time t	5	3	0.33	0	40
	20	3	0.33	0	40
	40	3	0.33	0	40

Table 6: Truncated Gaussian components of Mixture Model distribution for the Online CVRP. μ, σ, ω, a and b stand for mean, standard deviation, weight, minimum value and maximum value of each truncated Gaussian distribution correspondingly.

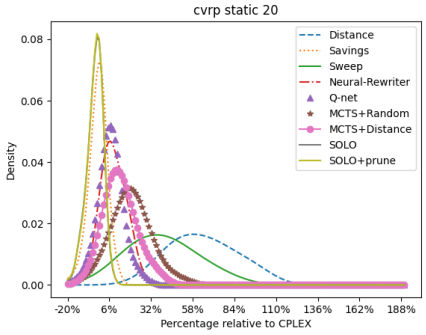

Figure 7: Offline CVRP with 20 customers.

In Figures 7-12 a deeper comparison between SOLO and other baselines can be found for the CVRP online and offline problems. Each line in the figures represents the distribution of relative rewards achieved by the baselines compared to the OR-Tools baseline (baselines with a negative mean value reached better mean results compared to OR-Tools). These figures are summarized in table 7. We see that in most cases our full method out preforms other algorithms and achieves good results.

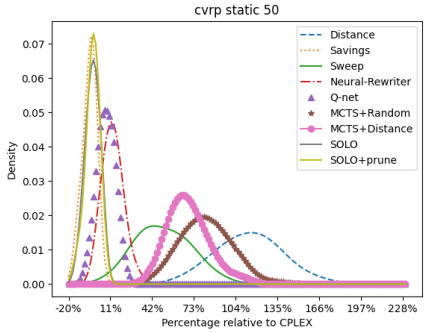

Figure 8: Offline CVRP with 50 customers.

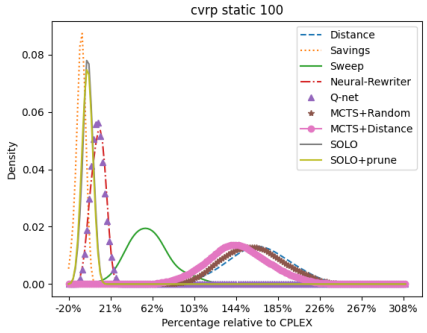

Figure 9: Offline CVRP with 100 customers.

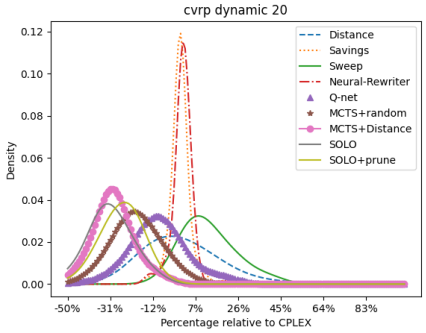

Figure 10: Online CVRP with 20 customers.

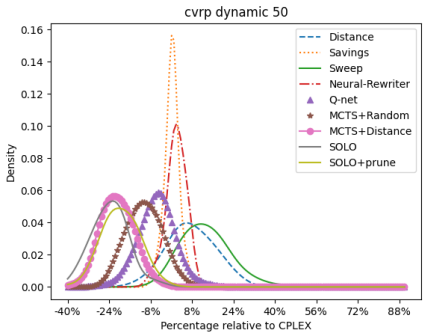

Figure 11: Online CVRP with 50 customers.

PMSP For all problems, we sample the job processing times, setup times, and job weights from the discrete uniform distributions: $U[1, 100]$, $U[1, 50]$ and $U[1, 10]$, respectively. Similar to the figures shown above, Figures 13 - 16 depict our comparisons between the SOLO and the various baselines, relative to the CPLEX baseline, for the PSMP Offline and Online problems (negative average indicates better results).

Runtime

As mentioned in Section 35 all benchmarks including SOLO were run using a timeout of 10 seconds. In SOLO, the time given limits the amount of rollouts MCTS can preform for each state. We can compute the worst-case runtime in seconds as $n_d \times 10$ with n_d being the number of decision points. For the Offline CVRP with disabled `noop` action (i.e., the agent is forced to move to some customer after it arrived to the depot) worst-case n_d is computed as $2 \times s_p$, where s_p is the problem size. This corresponds to the sub-optimal but yet acceptable strategy of routing back to the depot after every customer. For the Online setting worst-case n_d equals to $3 \times s_p$ and corresponds to waiting at the depot until all customers become visible and then following the sub-optimal offline strategy described above. Therefore the worst-case runtime will be 200, 500 and 1000 seconds for the problem sizes 20, 50 and 100 respectivly . Worst-case runtime of the Offline PMS will add up to 200 and 800 seconds for problems of size 20 and 80. For the Offline PMS n_d is equal to s_p since the `noop` action is disabled, meaning that at every decision point SOLO has to schedule one of the available jobs. For both the online PMS problems with different numbers of machines, n_d is equal to the expected number of arriving jobs that is set to 80.

The Advantage of Learning the Input Distribution

A well-studied problem in the field of online algorithms is the limiting effect of not having information about the arrival distribution (e.g., Garg, Gupta, and et al. 2008). In contrast, modern reinforcement learning algorithms often demonstrate a capacity to incorporate, either explicitly or implicitly, knowledge of the distribution (e.g., in their estimates of the Q values). In this context, we pose the following question:

Can an efficient optimal, or approximately optimal offline algorithm be naively adapted to the online version of the problem without regard for the arrivals distribution?

It is not hard to show that such adaptations generally do not preserve their offline optimality properties. To see that, consider the following examples for the two problems studied in this paper:

Online Capacitated Vehicle Routing: In this case, consider n arrivals from two distant locations p_1 and p_2 between time 0 and the horizon h. Each customer arrives at time $t \sim U[0, h]$ and is positioned at either location with prob-

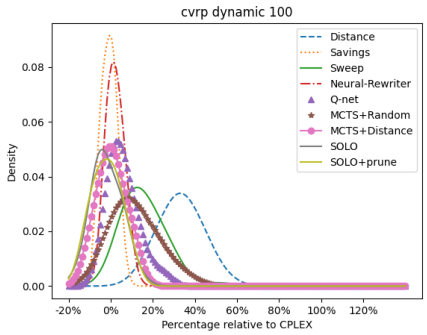

Figure 12: Online CVRP with 100 customers.

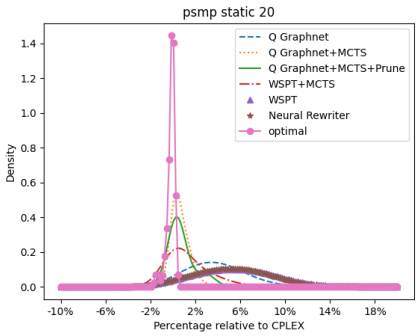

Figure 13: Offline PSMP with 20 jobs. Distribution of evaluations compared to CPLEX.

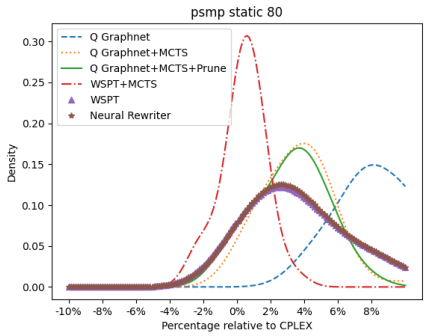

Figure 14: Offline PSMP with 80 jobs. Distribution of evaluations compared to CPLEX.

ability $\frac{1}{2}$, each. It can be shown that, in expectation, an offline optimal algorithm that is run in an online fashion would make $\Omega(n)$ travels between the two locations, whereas the algorithm that first handles customers at location p_1 and then moves on to location p_2 will make $O(1)$ such trips. As before, the expected ratio is thus unbounded.

Online Scheduling: Consider a simple distribution in which there are two job classes s_1 and s_2, where each incoming job belongs to either class with probability $\frac{1}{2}$. All weights and processing times are ϵ. The setup time incurred

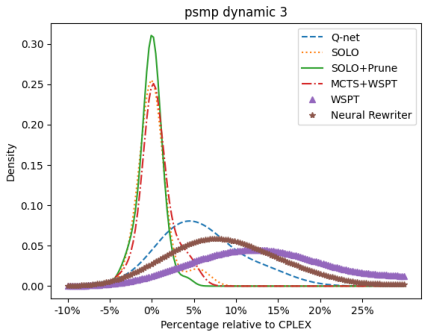

Figure 15: Online PSMP with 3 machines.

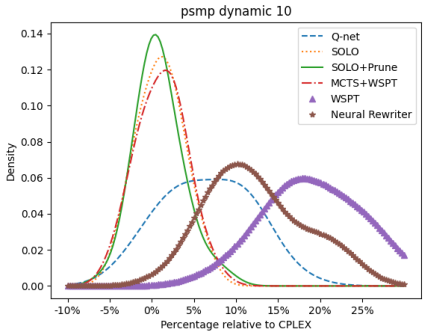

Figure 16: Online PSMP with 10 machines.

by switching from class s_1 to s_2 is 1, and the setup time incurred by switching from class s_2 to s_1 is again, ϵ. Now consider a minimal instantiation of the problem, in which there are two machines, and two batches of size 2, each. Then with probability $\frac{3}{16}$, the first batch will not contain class s_2 jobs, whereas the second batch will. In this case, an optimal offline distribution-agnostic algorithm will assign the two jobs to separate machines. On the other hand, a reasonable algorithm with access to the distribution will reserve a machine for future class s_2 jobs, due to its prohibitive setup costs, and put all s_1 jobs on the other machine. Since such instances occur with constant probability, it is easy to see that the online algorithm has a TWCT objective value of $O(\epsilon)$, while the offline algorithm's objective value will be $1 + O(\epsilon)$. Hence, the expected ratio of costs of the two algorithms will be unbounded as ϵ approaches 0.

	Offline CVRP		
	20	**50**	**100**
Uniform-Random[UR]	-13.21 (107.51%)	-30.58 (167.52%)	-58.84 (230.13%)
Distance[D]	-10.43 (63.65%)	-24.36 (113.39%)	-47.59 (167.38%)
Savings	-6.35 (-1.04%)	**-11.03 (-3.98%)**	**-16.51 (-7.94%)**
Sweep	-8.89 (39.33%)	-17.23 (49.94%)	-28.24 (58.11%)
OR-Tools	-6.42 (0.00%)	-11.50 (0.00%)	-17.96 (0.00%)
NeuralRewriter	-6.95 (8.48%)	-12.83 (11.83%)	-19.45 (8.57%)
Q-Net	-6.84 (6.59%)	-12.33 (7.37%)	-19.27 (7.62%)
MCTS+UR	-7.65 (19.45%)	-20.72 (81.08%)	-46.34 (160.11%)
MCTS+D	-7.15 (12.01%)	-18.91 (65.04%)	-44.00 (147.44%)
SOLO	**-6.21 (-3.18%)**	-11.25 (-1.98%)	-17.68 (-1.24%)
	Online CVRP		
	20	**50**	**100**
Uniform-Random[UR]	-12.72 (31.67%)	-28.00 (54.25%)	-52.73 (108.06%)
Distance[D]	-9.75 (0.76%)	-19.70 (8.48%)	-33.65 (32.72%)
Savings	-9.90 (0.51%)	-18.19 (0.07%)	-25.15 (-0.90%)
Sweep	-11.16 (13.73%)	-20.75 (14.27%)	-29.52 (16.16%)
OR-Tools	-9.86 (0.00%)	-18.18 (0.00%)	-25.40 (0.00%)
NeuralRewriter	-10.00 (1.56%)	-18.54 (2.00%)	-25.85 (1.90%)
Q-net	-8.79 (-9.76%)	-17.26 (-4.99%)	-26.80 (5.70%)
MCTS+UR	-7.80 (-20.27%)	-16.54 (-9.06%)	-28.72 (12.98%)
MCTS+D	-6.78 (-30.84%)	-14.59 (-19.74%)	-25.58 (0.78%)
SOLO	**-6.63 (-32.38%)**	**-14.03 (-22.82%)**	**-24.80 (-2.28%)**

Table 7: Offline and Online CVRP results. Each cell contains the average cost and the fractional improvements over OR-Tools (negative numbers are better than OR-Tools). Best results and our methods are marked in bold