A State Aggregation Approach for Solving Knapsack Problem with Deep Reinforcement Learning

A State Aggregation Approach for Solving Knapsack Problem with Deep Reinforcement Learning

Abstract

This paper proposes a Deep Reinforcement Learning (DRL) approach for solving knapsack problem. The proposed method consists of a state aggregation step based on tabular reinforcement learning to extract features and construct states. The state aggregation policy is applied to each problem instance of the knapsack problem, which is used with Advantage Actor Critic (A2C) algorithm to train a policy through which the items are sequentially selected at each time step. The method is a constructive solution approach and the process of selecting items is repeated until the final solution is obtained. The experiments show that our approach provides close to optimal solutions for all tested instances, outperforms the greedy algorithm, and is able to handle larger instances and more flexible than an existing DRL approach. In addition, the results demonstrate that the proposed model with the state aggregation strategy not only gives better solutions but also learns in less timesteps, than the one without state aggregation.

Keywords: Combinatorial Optimization Problems, Knapsack Problem, Deep Reinforcement Learning, State Aggregation.

1. Introduction

Heuristic algorithms for solving Combinatorial Optimization Problems (COPs) achieve acceptable solutions in a polynomial time. These algorithms rely on handcrafted heuristics that conduct the process of finding the solutions. Although these heuristics work well in many COPs, they mostly rely on the nature of problems and they need to be revised for different problem statements (Bello et al., 2017). In this paper, we aim to learn and improve the handcrafted heuristics to improve the quality of the solutions. We study *knapsack problem (KP)*, which is one of the well-known benchmark problems in COPs. KP is defined as selecting some items from a given set such that the selected items fit in the knapsack and their total value is maximized. This problem has many applications such as cargo loading, cutting stock and capital budgeting (Wilbaut et al., 2008).

Recently, there is a great progress in Artificial Intelligence (AI) literature in developing machine learning (ML) methods to solve COPs (Bengio et al., 2018), where a promising ML based method is *Deep Reinforcement Learning (DRL)*. DRL is the integration of *Reinforcement Learning (RL)* and *Deep Neural Networks (DNN)* (Arulkumaran et al., 2017; LeCun et al., 1998; Huang et al., 2019). Several DRL based approaches have been proposed to solve the Traveling Salesman Problem (TSP), e.g. (Bello et al., 2017; Joshi et al., 2019; Kool et al., 2019) that mainly use sequence to sequence modeling because the solution of a TSP is a sequence, i.e. a permutation of the input. These approaches work well for the TSP problem, however, in the Knapsack problem, the solutions are subsets of the inputs. Hence, the existing DRL based approaches for solving sequence to sequence problems like

TSP might not work well for KP. Furthermore, KP solutions require to be feasible and satisfy constraints which are different with TSP. Bello et al. (2017) solve a Knapsack problem using the policy gradient algorithm with pointer networks. Although their method solves instances up to 200 items optimally, the following limitations are identified: (1) intractability to large instances: the state space grows rapidly with increasing number of items, and (2) generality to other sizes of instances: the trained model is applicable for solving the problems that have exactly the same knapsack capacity and the same number of items. In this paper, we introduce a DRL approach with state aggregation that boosts the capability of the typical greedy algorithm and improves this heuristic. Besides, our method does not have the aforementioned limitations.

We propose a state aggregation method to discretize the feature values of items. A tabular reinforcement learning is used to learn the best aggregation strategy for each item. This discretized features not only provide a discrete representation of the problem instances, but also reduces the state space by reducing the number of unique values. Since even the reduced state space after applying the state aggregation is still large, DRL is employed as a powerful function approximation method. We use *Advantage Actor Critic (A2C)* algorithm to learn the policy of selecting items. A2C makes use of two DNNs for learning policy and value functions (Mnih et al., 2016). The policy DNN has an output size that is equal to the number of items in the KP instance. The proposed method greedily solves the KP problem by successive item selections and placing them in the knapsack, each is done by following a greedy or softmax algorithm on the output of the policy DNN.

The experimental results show that the proposed approach finds optimal solutions for the problem instances of size 50 that is used in Bello et al. (2017). Moreover, we show the method obtains close to optimal solutions for three different types of instances with at most 50, 300 and 500 items. We also demonstrate that the proposed DRL method with state aggregation performs better than the DRL without aggregation in terms of both learning rate and the solution quality. Finally, we compare the results with two approaches based on pointer network (Bello et al., 2017; Gu and Hao, 2018). Although many works focus on sequence to sequence problems like TSP, they are not directly applicable on KPs. We summarize our contributions as follows.

- Our DRL-based approach to solve KP improves the heuristic greedy algorithm for 0-1 KP and shows better performance than the existing DRL approaches. The developed method can be trained once for N items and it can be used for any KP instances with size up to N.

- Developing a state aggregation strategy to derive state embedding that reduces the state space size. This general strategy effectively speed up learning on solving KP.

2. Related Work

It may take exponential time, in the size of instances, to solve most of COPs optimally due to their NP-Hardness (Karp, 1972; Cook, 2006). Knapsack Problem (KP) has gained a remarkable attention in the literature. Despite the fact that the fractional KP is optimally solvable by the heuristic greedy algorithm, the 0-1 knapsack problem is NP-Hard (Cormen et al., 2009), and a large variety of KPs remain hard to solve (Pisinger, 2005). Moreover,

it has been shown by empirical evidence that solving instances near the phase transition are challenging for humans (Yadav et al., 2018). The phase transition emerges around critical values of items and capacity so that the probability of having a solution for an instance changes from zero to one. Many algorithms, ranging from dynamic programming algorithms, e.g. (Dasgupta et al., 2008), to meta-heuristics, e.g. (Feng et al., 2018) have been proposed to solve KP.

Cleverly searching and branch and bound methods can prune the search tree and reduce computational times for solving COPs in practice (Woeginger, 2003). However, these methods are still prohibitive for large instances. Although polynomial time approximation schemes and integer linear programming (ILP) based approaches might be performed in reasonable time, they rely on handcraft heuristics and they may suffer from weak optimality (Vazirani, 2013; Du and Pardalos, 2013). In order to cope with this limitations, Machine Learning (ML) based and data driven methods are developed to learn heuristics.

In (Vinyals et al., 2015), the Pointer Network architecture is introduced where the output layer of the DNN is a function of the input. In (Bello et al., 2017), the pointer network is used with RL to solve the Traveling Salesman Problem (TSP). They use policy gradient and a variant of Asynchronous Advantage Actor-Critic (A3C) algorithm of Mnih et al. (2016) to train a DNN, and show close to optimal solutions are found for up to 100 cities. In (Khalil et al., 2017) another neural network framework is introduced for graph-based COPs, where *structure2vec* of Dai et al. (2016) is used to derive an embedding for the vertices of the graph. The structure2vec computes a p-dimensional feature embedding for each node and a parametric Q function is trained using Q learning algorithm. In (Kool et al., 2019), the pointer network is incorporated with attention layers. With the REINFORCE algorithm, they obtained close to optimal solutions for the TSP instances of up to 100 nodes.

Most of ML-based research on solving COPs focuses on TSP. COPs like TSP and Vehicle Routing Problem that have gained high attentions in past few years, require a sequence of the input as the solution and sequence-to-sequence neural architectures might be proper approaches for solving them (Sutskever et al., 2014). However, the solutions of COPs like KP are a subset of the input. This issue makes the original sequence-to-sequence approaches inapplicable for solving KP. Recently, a pointer network deep learning approach is presented for solving 0-1 KP (Gu and Hao, 2018). This method is based on supervise learning and optimal solutions which is not available in most of the cases. In this paper we propose a DRL framework for subset selection problems.

3. Problem Definition and Modeling

An instance of *0-1 Knapsack Problem*, denoted by P, includes a set $\mathcal{I}_\mathcal{P}$ of items and a knapsack with capacity W_P. For practical reasons in the notation of further sections, we have $|\mathcal{I}_\mathcal{P}| = n_P$. Each item $i \in \mathcal{I}_\mathcal{P}$ has value v_i and weight w_i. The goal is to maximize the total value of a selected subset of items such that the total weight of the selected items does not exceed the knapsack capacity W_P. Since P is a 0-1 KP, selecting a fraction of an item is not possible.

Our method for solving this variant of KP is based on deep reinforcement learning. We assume that the number of items is variable and a constructive solution can solve the problem by considering the capacity constraint. Therefore, the process of selecting a subset

of items $\mathcal{I}'_{\mathcal{P}} \subseteq \mathcal{I}_{\mathcal{P}}$ is modeled as a sequential decision process. The policy DNN is trained with A2C introduced in (Mnih et al., 2016) on a set of problem instances with at most N items. The information of each problem instance consists of $|\mathcal{I}_{\mathcal{P}}| = n_P \leq N$ items with value v_i and weight w_i for each $i \in \mathcal{I}_{\mathcal{P}}$ and together with W_P, they are the inputs of DNN. The DNN has N outputs that each being associated with a value of selecting a specific item $i \in \mathcal{I}_{\mathcal{P}}$. The policy is to select the item with highest selection value in each step. After selecting item i, it is removed from the original problem instance P and a new problem instance P' with a reduced item set $\mathcal{I}_{\mathcal{P}'} = \mathcal{I}_{\mathcal{P}} \setminus \{i\}$ and capacity $W_{P'} = W_P - w_i$ is generated. For the cases where i cannot be added to the knapsack because of the capacity constraint, the new instance P' is generated by simply removing i from the item set, without altering W_P. In this way, when the policy is trained with problem instances of at most N items, it can be used to find solutions for new instances as long as their item sizes are no greater than N.

Such KP problems can be found in different applications. For example, an online ad publisher faces with a set of advertisements. Assuming a fixed upper bound for the number of ads, the problem is to select a subset of them to show to the users. In this example, the values are relevance scores and the weights are the size of ads banners. The goal is to fill a slot of a certain size with the ads.

4. DRL-based KP Solver

Figure 1 shows the overview of our proposed method. It consists of two components. The first component includes a formulation of KP to MDP, which is solved using a DRL approach (Algorithm 1). The second component is a state aggregation method (Algorithm 2), which learns an aggregation policy to discretize states that serve as inputs to DRL.

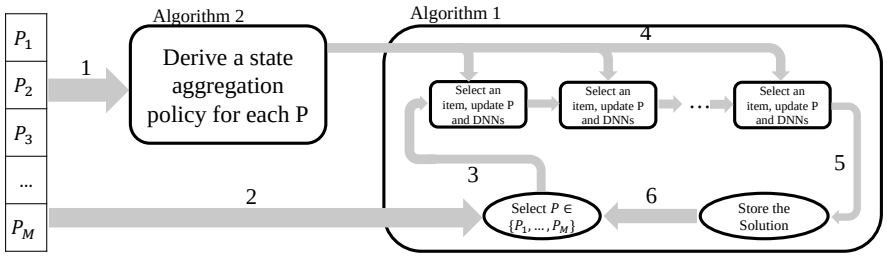


Figure 1: The overview of the KP solver. 1) A set of instances are used for deriving an aggregation policy for items. 2) The same set of problems are used with DRL. 3) A problem instance is selected for training. 4) Items are selected sequentially until finding a solution. At each step the updated P is aggregated to find the state. The parameters of value and policy DNNs are updated using A2C. 5) The best solution is stored. 6) Another problem instance is selected for training. The process continues for a certain number of timesteps.

4.1 Deep Reinforcement Learning method

In order to solve the 0-1 KP, DRL is used to derive a policy through that the items are sequentially added to the solution. We define the states, actions and rewards of DRL modeling of KPs for an instance P' which is a representation of P after selecting some items, as follows.

States $s(P)$: A complete set of information of instance P' containing $n_{P'}$, v_i and w_i for $n_{P'}$ items, capacity $W_{P'}$, the total value of the items ($Sv = \sum_{i \in \mathcal{I}_{P'}} v_i$), and the total weight of the items ($Sw = \sum_{i \in \mathcal{I}_{P'}} w_i$) makes a feature vector of $2n_{P'} + 4$ features. Since $n_{P'} \leq N$ for all P', the feature vector of the instances that have $n_{P'} < N$ items consists of $2N + 4$ features such that the first $2n_{P'} + 4$ features carry the information of the problem instance and the remaining ones are zero. Section 4.2 will reduce this feature vector by a state aggregation strategy.

Actions: There are N actions $A_1, A_2, ..., A_N$, each corresponding to select one item. At each decision moment, a state is fed to the policy DNN and an action is selected according to the output of the DNN.

Reward Function: The reward function is defined based on three criteria. First, if item i can be added to the knapsack without exceeding the capacity limit, the reward is positive. Second, if w_i is greater than $W_{P'}$, i.e. i cannot be added to the knapsack, the reward is negative. Third, for each instance P' where $n_{P'} < N$, the first $n_{P'}$ outputs of DNN correspond to the items of P' and the next $N - n_{P'}$ outputs are undefined actions because the corresponding items do not exist. Therefore, a large penalty i.e. $-W_{P'}$ is used for the reward of choosing undefined actions. We separate the reward of undefined action and heavy items because an action with $i > n$ is always undefined, however items with $w_i > W_P$ could be added to the knapsack if they were selected in earlier steps. Therefore, their penalty is lower. The normalized values vr_i and normalized weights wr_i which are explained in Section 4.2 are used as positive and negative rewards respectively. Equation (1) shows the reward of state $s(P')$ and action A_i.

$$r(s(P'), A_i) = \begin{cases} -W_{P'} & \text{if } \ i > n_{P'} \\ vr_i & \text{if } \ w_i \leq W_{P'} \\ -wr_i & \text{if } \ w_i > W_{P'} \end{cases} \tag{1}$$

Employing these definitions of states, actions and rewards, the A2C algorithm is used for training policy and value DNNs (Mnih et al., 2016), where two DNNs are used for policy (π) and value (V) functions respectively. The advantage value is obtained by subtracting state values (V) from state action values (Q) which is defined by $r + \gamma V(s_{t+1})$. This value is used in gradient function to update the parameters of the DNNs using Equations (2) and (3) (Mnih et al., 2016; Hill et al., 2018).

$$\theta^{t+1} \leftarrow \theta^t + \nabla_{\theta^t} \log \pi(A_i | s(P), \theta^t)[r_t + \gamma V(s(P'), \theta_v^t) - V(s(P), \theta_v^t)] \tag{2}$$

$$\theta_v^{t+1} \leftarrow \theta_v^t + \frac{\partial (r_t + \gamma V(s(P'), \theta_v^t) - V(s(P), \theta_v^t))^2}{\partial \theta_v^t} \tag{3}$$

Algorithm 1 DRL-based Knapsack Solver

Input: M Problem Instances each having at most N items
Output: Values of solutions of the M instances

Initialize a policy network $\pi(A_i|s,\theta)$ with $2N+4$ inputs, N outputs and parameters θ.
Initialize a value DNN with parameters θ_v as $V(s,\theta_v)$
$t_{max} = 3N \times 10^4$, $t = 0$
Initialize Val: a list of length M, all 0
while $t < t_{max}$ **do**
  Select a problem instance P with capacity W_P.
  $ow = 0$ {Total weight of selected items}
  $ov = 0$ {Total values of selected items}
  $P' \leftarrow P$, $n_{P'} \leftarrow n_P$, $W_{P'} \leftarrow W_P$
  while $ow < W_{P'}$ and $n_{P'} > 0$ **do**
    Find $s(P')$ using state aggregation strategy (Eqn. (7))
    Perform action i according to policy $\pi(A_i|s(P'),\theta^t)$ and observe $r(s(P'),A_i)$
    if $i \leq n_{P'}$ and $w_i + ow \leq W_{P'}$ **then**
      $ow \leftarrow ow + w_i$, $ov \leftarrow ov + v_i$, $W_{P'} \leftarrow W_{P'} - w_i$
    end if
    $P' \leftarrow P' \setminus \{i\}$, $n_{P'} \leftarrow n_{P'} - 1$
    Update θ and θ_v using Eqns. (2) and (3)
    $t \leftarrow t + 1$
  end while
  if $ov > Val[P']$ **then**
    $Val[P'] \leftarrow ov$
  end if
end while
return Val

where, θ^t and θ_v^t are the parameters of policy and value DNNs in decision moment t respectively. The corresponding state of a problem instance P is fed to the policy DNN and the items can be selected by following a policy according to the output of the policy DNN. Upon selecting an item, P' is obtained from P and it is again fed to the policy DNN to select the next item. This process is continued until filling the knapsack or exceeding the weight constraint. Algorithm 1 shows the DRL-based knapsack solver method.

4.2 State Aggregation

As the number of items increases, the state space grows up exponentially and this affects the performance of function approximation with DNN. In order to shrink the state space and boost the method to have the capability of solving large problem instances, a new state embedding is derived by state aggregation. Specifically, the problem is to find a certain number of split points on the values of items and transform the values into integers using these split points. In other words, the feature values of states are divided into subsets and the values of each subset are converted to a certain value. Finding proper number of split points is difficult because the evaluation should be performed after training the model with the defined states which is computationally expensive. Besides, the number of split points for each item influence on the number of split points of other items because the objective is to decrease the total number of states entailed by transforming all items. Therefore, we

opt for reinforcement learning to tackle delayed reward and to sequentially determine the proper number of split points for each item. The problem instances are used to model the environment which is explained in the next section.

Preparing data. A set of problem instances are used for deriving the state embedding. Each problem instance is identified by a set of feature values which are the items information and capacity. The first step in aggregating the states is to generate random solutions for each problem instance. As mentioned before, an episode is a sequence of states and actions that each action selects an item and the solution is the set of selected items. These instances can be shown in a table in which each row corresponds to a problem instance and the columns are items information.

One issue in selecting the feature vector of original items information as states is that different KP instances are not comparable because the values and weights of items might be very different. As an example, assume that values and weights of an instance are integer numbers between 1 to 10, while these values and weights lies between 100 and 110 for another instance. Generalization based on these different values is difficult, although their ratio are similar. In order to solve this issue, for each item of instance P, all v_i are normalized through dividing by the product of w_i and W_P as shown in Eqn. (4). Furthermore, the ratio between w_i and W is also calculated based on Eqn. (5). The v_i and w_i for each item are replaced with these two ratios in the feature vector of P. This modification makes the items of different problems comparable. The learned policy network in this way would boost the capability of the well-known heuristic greedy algorithm which is optimal for fractional KP.

$$vr_i(v_i, w_i, W_P) = \frac{v_i}{w_i \times W_P} \tag{4}$$

$$wr_i(v_i, w_i, W_P) = \frac{w_i}{W_P} \tag{5}$$

where, vr_i and wr_i are the normalized value and normalized weight respectively. For a problem instance P, vr_i, wr_i, W_P, Sv and Sw construct a feature vector $F(P) = (F_1, ..., F_{2N+4}) = (n_P, W_P, Sv, Sw, vr_1, wr_1, ..., vr_{n_P}, wr_{n_P})$, where $F(P)$ is the feature vector of P, Sv and Sw are the sum of remaining values and weights respectively.

After obtaining a table of problem instances with comparable items, we sort for each row (i.e. each problem instance) the columns (i.e. vr_i and wr_i) in descending order with respect to vr_i. In other words, the first two columns of each row, i.e. vr_1 and wr_1 correspond to the item with the highest normalized value. The second two columns which are vr_2 and wr_2, correspond to the item with the second highest normalized value and so on. For a problem P with $n_P < N$, the items information are located from the columns vr_1 and wr_1 to vr_{n_P} and wr_{n_P} respectively. The values of vr_{n_P+1} to vr_N and also the values of wr_{n_P+1} to wr_N are zero. This ordering helps to aggregate all the highest vr_i of all problem instances with a single aggregation strategy because the problem instances are comparable and the highest vr_i is in a certain column. This explanation holds for second, third and other highest vr_i. Each column is called a feature and the next step is to derive an aggregation strategy for the values of each feature.

State aggregation through Q-Learning. The idea of the aggregation is to reduce the number of unique values for all features. We do such reduction by splitting the values of

one feature into several groups and then mapping each group's value to a particular integer. The proper number of splits for each feature is learned by reinforcement learning. For each feature F_k that $k \in \{1, ..., 2N + 4\}$, let action d_{F_k} be the number of splits on the values of the feature F_k, and $F_{k,P}$ be the value of feature F_k for problem instance P. Among all features, we perform state aggregation on vr_i and wr_i of item i.

For aggregating the values of vr_i of all M problem instances, action $d_{vr_i} \in \{1, 2, ..., x\}$ is the number of splits where its optimal value i.e. $d^*_{vr_i}$ is obtained by Algorithm 2. Using $d^*_{vr_i}$ splits, the values of vr_i are divided into $d^*_{vr_i} + 1$ subsets and all the subsets except the last one have $\lceil \frac{M}{d^*_{vr_i}+1} \rceil$ values. The last subset has $M - (\lceil \frac{M}{d^*_{vr_i}+1} \rceil d^*_{vr_i})$ values. Then, all values of each subset is converted to an integer starting from 0. This process transforms the values of feature vr_i to a set of integers $\{0, 1, ..., d^*_{vr_i}\}$. As an example, assume there are $M = 7$ problem instances that the values of vr_1 are $(1, 2, 6, 3, 1, 2, 5)$ and $d^*_{vr_1}$ is 2. These values need to be divided into $d^*_{vr_1} + 1 = 3$ subsets. First the sorted values $(1, 1, 2, 2, 3, 5, 6)$ are acquired. Then, three subsets $(\{1, 1, 2\}, \{2, 3, 5\}, \{6\})$ are obtained that each has $\lceil 7/3 \rceil = 3$ values except the last one which has one value. Finally, the values of vr_1 are aggregated and the new values are $(0, 0, 2, 1, 1, 0, 1)$. The aggregation reduces 5 unique values of vr_1 to 3 unique values.

For all wr_i, $d^*_{wr_i}$ is 2 and the split points are 0.5 and 1. The motivation of this hard setting is separating illegal, light and heavy weights. Illegal weights are the weights with $wr_i > 1$ that cannot be added to the knapsack. Similarly, $wr_i \leq 0.5$ and $0.5 < wr_i \leq 1$ determine light and heavy items respectively. The aggregation process is performed by the function $map(F_{k,P}, d^*_{F_k})$ that gets $F_{k,P}$ and returns an integer which corresponds to a subset based on $d^*_{F_k}$ splits.

We use heuristics to define the reward function $R(F_k, d_{F_k})$ which is shown in (6).

$$R(F_k, d_{F_k}) = \frac{\prod_{j=1}^{d_{F_k}+1} l_{F_k,j}}{(d_{F_k} + 1)c_{F_k,d_{F_k}}} \tag{6}$$

where, $l_{F_k,j}$ is the size of j^{th} subset, and $c_{F_k,d_{F_k}}$ is the number of all common values between all subsets. Three main motivations of designing rewards are: (1) We aim to define the reward function such that it reduces the size of state space. The number of unique states for each feature is $d_{F_k} + 1$ after applying d_{F_k} splits and this value inversely relates to the reward of each action; (2) For feature F_k and d_{F_k} splits, let $j \in \{0, 1, 2, ..., d_{F_k}\}$ be a subset based on d_{F_k} splits and $l_{F_k,j}$ be the difference between maximum and minimum values of j^{th} subset. As larger $l_{F_k,j}$ entail in aggregating more values, their rewards are higher than those for smaller $l_{F_k,j}$. However, unequal subsets contain unequal number of values. For example, if the feature values are uniformly dispersed between 0 and 10, creating two subsets with lengths 5 and 5 are better than two subsets with length 1 and 9. Therefore, the product of the $l_{F_k,j}$ for all j is in the numerator of the reward function; and (3) Distinct states help an agent to derive a deterministic policy because states have dissimilar features. Likewise, two subsets with less overlapped values represent different sets of states and the policy can better distinguish them. For example, for the subsets $(\{1, 1, 2\}, \{2, 3, 5\}, \{6\})$, 2 is common between two subsets and it can be assigned to both subsets. Assigning this value to different subsets entails a different policy that may have different performance. In order to reduce

the number of common values between two groups, we define $c_{F_k,d_{F_k}}$ as the total number of common values in different subsets.

A Q table is constructed for the states and actions and it is filled by the $Q-$learning algorithm (Sutton et al., 1998) as shown in Algorithm 2. Each vr_i is a state and the next state is the $vr_{i'}$ which i' is an arbitrary state. Finally, an optimal decision is found by using the Q table for each feature. The algorithm is used for aggregating vr_i and we denote $d^*_{vr_i}$ as the optimal aggregation action for each vr_i. The state embedding derived by this strategy is a feature vector consisting of aggregated features and this state embedding is used in line 11 of algorithm 1. Equation (7) shows $s(P)$, the state embedding of P.

$$s(P) = \{map(F_{k,P}, d^*_{F_k}) : \quad \forall F_k \in F(P)\} \tag{7}$$

Algorithm 2 Q-Learning for State Aggregation

Input: Feature table of problem instances $P_1, ..., P_M$
Output: The number of optimal split points for all vr_i

1: Initialize a Q table with N rows and x columns. States are features and actions are the number of split points
2: Select item i randomly
3: **repeat**
4: Select i' randomly as the next item
5: Select $d_{vr_i} \in \{1, ..., x\}$ according to ϵ-greedy policy
6: Find $R(vr_i, d_{vr_i})$ using Eqn. 6
7: Update $Q(vr_i, d_{vr_i}) \leftarrow Q(vr_i, d_{vr_i}) + \alpha[R_{vr_i,d_{vr_i}} + \gamma \max_{d'} Q(vr_{i'}, d') - Q(vr_i, d_{vr_i})]$
8: $i = i'$
9: **until** Convergence
10: **return** $d^*_{vr_i} = argmax_d Q(vr_i, d)\ \forall i$

5. Experiments

The proposed DRL with aggregation algorithm is compared with (1) greedy algorithm and (2) DRL without aggregation (3) DRL approaches for solving KPs with pointer network (Bello et al., 2017) (4) Pointer Network and Supervised learning method (Gu and Hao, 2018). There are several works using DRL for COPs. However, they are mainly for sequence to sequence problems like TSP and they are not directly applicable on KPs. The problem instances and code used for experiments are available in URL[1].

5.1 Configuration of the method

We first choose the DRL algorithm for training the policy DNN by testing three algorithms: Deep Q Network (DQN) (Mnih et al., 2013), Advantage Actor Critic (A2C) (Mnih et al., 2016), and Proximal Policy Optimization (Schulman et al., 2017). We use *stable-baseline* tools to implement the A2C algorithm (Hill et al., 2018). We test the algorithms with different parameters on a set of knapsack problem instances with at most 50 items. The policy and the value networks consist of two layers of 64 nodes where the *sigmoid* function is

1. The link is not shown due to the blind review.

Table 1: The performance of three DRL algorithms and reward functions for solving 1000 randomly generated KPs with at most 50 items. Each column corresponds to a reward definition for positive and negative rewards shown by + and -. The numbers in the parentheses denote the average solution value and the number of optimally solved instances respectively. The results are obtained by averaging over ten separate runs each for 10^4 episodes for aggregated states. + and − in Reward column denote the rewards corresponding to $w_i \leq W_{P'}$ and $w_i > W_{P'}$ respectively.

Algorithm	$+ : vr_i, - : wr_i$	$+ : vr_i, - : w_i$	$+ : v_i, - : wr_i$	$+ : v_i, - : w_i$	+: +1, -: -1
A2C	**(434.50, 959)**	(433.77, 881)	(433.68, 882)	(433.47, 854)	(432.63, 802)
DQN	(367.88, 384)	(405.14, 501)	(432.62, 794)	(432.96, 816)	(369.92, 382)
PPO	(431.58, 725)	(431.76, 737)	(431.3, 714)	(432.05, 762)	(431.08, 697)

used in the output layer and *Adam Optimizer* is employed for optimizing the weights. The learning rate is 0.001 and the weights are updated after five steps. Since the number of items are finite, a close to one discount factor is chosen. The method is trained on 10^4 episodes which are selected from M instances. We also consider v_i and vr_i for positive rewards, and w_i and wr_i for negative rewards and run the method ten times on all combinations of positive and negative rewards to infer the best definition of the reward function. The positive and the negative values are selected based on values and weights because they directly define the profit and constraints of the problem. Furthermore, we use constant values +1 and -1 for positive and negative rewards respectively to show that item-dependent rewards perform better. Table 1 shows the performance of these three DRL algorithms in terms of the solution quality and number of optimally solved instances. The table shows that using vr_i and wr_i for rewards, A2C provides a higher solution value and more optimally solved instances than the other algorithms. The reason that A2C works better than PPO is because the difference between close to optimal solutions is very low and the *CLIP* in PPO discards this difference. A2C works better than DQN because the off-policy nature and $\epsilon - greedy$ algorithm which is used in DQN prevent the method from exploring close to the current-best solution. These results are found based on averaging over ten separate runs to show the stability of the results.

Based on the results, A2C is selected to be used in the remaining experiments to train the policy DNN in our approach.

5.2 Problem Instances

We use three different types of instances in the experiments: *Random Instances*, *Fixed W_P Instances* and *Hard Instances*. A set of M problem instances makes a dataset that the maximum number of items over all instances in the dataset is N.

Random instances (RI): A dataset of random instances has M problem instances that each instance P has $n_P \in \{1, 2, ..., N\}$ items. For an item i, v_i and w_i are randomly generated integers from one to R that is a fixed upper bound for v_i and w_i. The W_P is a random integer between $R/10$ and $3R$. Three datasets of random instances are generated with $M = 1000$, $N = 50, 300, 500$, and $R = 100, 600, 1800$, respectively.

Fixed W_P Instances (FI): In (Bello et al., 2017) a set of KP instances with fixed capacity and fixed item set size are used for evaluation. We generated three datasets of the same instances with $M = 1000$. The N for these three datasets is 50, 300 and 500 respectively. The values and the weights of all items in the three datasets are random real numbers between zero and one. The W_P is fixed for all the instances and it is 12.5 for $N = 50$, 37.5 for $N = 300$ and 37.5 for $N = 500$.
Hard instances (HI): In (Pisinger, 2005), a group of hard to solve problem instances were introduced that for each item i, v_i is strongly correlated with w_i. Specifically, w_i is a random integer in $[1, R]$, $v_i = w_i + R/10$ and $W_P = \frac{p}{M+1}\Sigma_{i=1}^{n_P} w_i$ where p is the id of P. Three datasets of $M = 1000$ hard instances are generated. For the first dataset, N is 50 and R is 100. Likewise, N is 300 and 500, and R is 600 and 1000 for the second and the third datasets respectively.

5.3 Evaluation Metrics

We evaluate the performance of different approaches in terms of the following metrics.
Average values of solutions ($\overline{Val}$). For each dataset of M problem instances, $\overline{Val}$ is the average of all solution values (total values of the selected items). Likewise, $\overline{Val}_{opt}$ is the average values of optimal solutions, which are obtained using the optimization solver *Gurobi* (Gurobi Optimization, 2020).
Learning rate. To compare how fast DRL with and without aggregation methods learn, the rate of increasing in Val is calculated per timesteps and the result is shown for each instance type when $N = 300$.
Number of optimally solved instances ($\#_{opt}$). In order to evaluate the performance of the method on the individual problem instances, the number of instances that the method finds their optimal solution is computed for each dataset.
Number of instances with highest solution value ($\#_{highest}$). This metric compares the solution values of (1) DRL with aggregation; (2) DRL without aggregation; and (3) greedy algorithm. It counts the number of wins each algorithm has in terms of solution values.

5.4 Experiment results

Table 2 shows the quality of the solutions of Greedy, DRL algorithms with (i.e. w/) and without (i.e. w/o) aggregation in solving three types of KP instances: RI, FI, and HI. The column $\overline{Val}_{opt}$ contains the average value of optimal solutions. Table 2 contains the ratio of $\overline{Val}$ and $\overline{Val}_{opt}$. These values show that the ratios of the solutions provided by our proposed method (DRL w/ aggregation) and the optimal solutions are most of the times more than 99.9%. This ratio does not change considerably when the number of items increases. Hence we conclude that our DRL based approach is able to find very close to optimal solutions for all instances we tested.

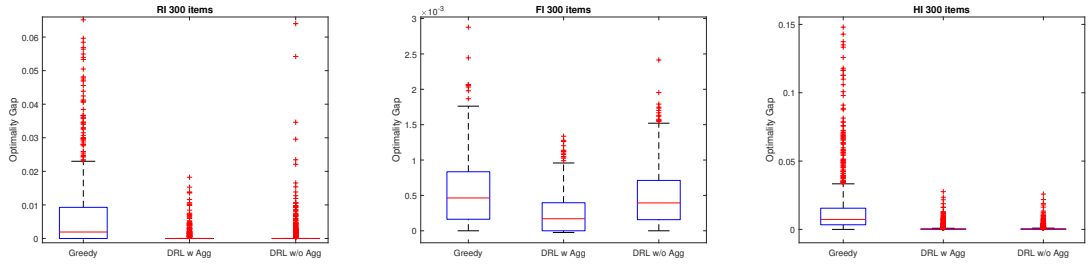
Figure 2 shows the box plots of optimality gap for the solutions of different instance types for 300 and 500 items. The optimality gap for each instance P is $1 - \frac{Val_A^P}{Val_{opt}}$ where Val_A^P is the solution value of running algorithm A on instance P, and Val_{opt} is the optimal solution value of P. The difference between the solutions of greedy algorithm and DRL algorithms can be better observed in this figure. Furthermore, the solutions of DRL algorithtms are

Table 2: Results of different algorithms and datasets of $M = 1000$ problem instances. It is possible that two approaches find the optimal solution for a certain instance. Hence, the total number.

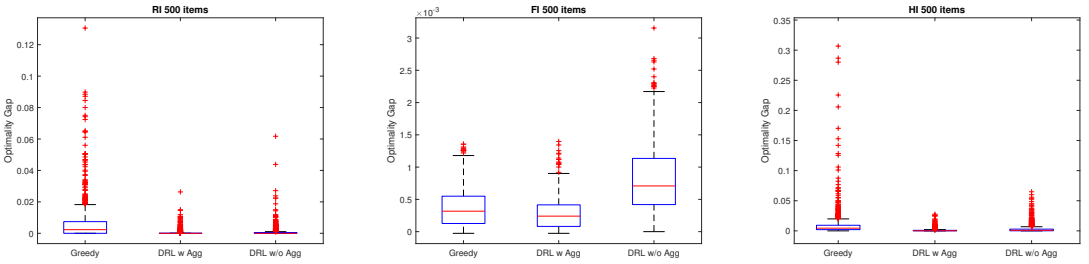
Dataset	Method	N	$\overline{Val}$	$\#_{opt}$	$\#_{highest}$	$\overline{Val}_{opt}$	$\frac{\overline{Val}}{\overline{Val}_{opt}}$
RI	Greedy	50	429.10	596	0	434.78	98.694%
	DRL w/o aggregation	50	434.09	893	7	434.78	99.843%
	DRL w/ aggregation	50	**434.50**	**959**	**41**	434.78	99.937%
	Greedy	300	1144.96	418	0	1151.58	99.425%
	DRL w/o aggregation	300	1150.83	830	21	1151.58	99.934%
	DRL w/ aggregation	300	**1151.10**	**878**	**47**	1151.58	99.958%
	Greedy	500	15216.51	345	0	15285.56	99.548%
	DRL w/o aggregation	500	15273.47	701	30	15285.56	99.920%
	DRL w/ aggregation	500	**15278.44**	**786**	**80**	15285.56	99.953%
FI	Greedy	50	20.10	172	0	20.15	99.738%
	DRL w/o aggregation	50	20.14	740	36	20.15	99.931%
	DRL w/ aggregation	50	**20.15**	**773**	**54**	20.15	99.959%
	Greedy	300	86.26	202	0	86.31	99.942%
	DRL w/o aggregation	300	86.27	226	24	86.31	99.961%
	DRL w/ aggregation	300	**86.29**	**330**	**205**	86.31	99.976%
	Greedy	500	111.68	204	0	111.73	99.945%
	DRL w/o aggregation	500	111.63	64	31	111.73	99.871%
	DRL w/ aggregation	500	**111.70**	**261**	**144**	111.73	99.970%
HI	Greedy	50	772.428	134	0	802.72	96.226%
	DRL w/o aggregation	50	799.036	655	113	802.72	99.540%
	DRL w/ aggregation	50	**799.438**	**689**	**147**	802.72	99.591%
	Greedy	300	27778.03	37	0	27965.76	99.328%
	DRL w/o aggregation	300	27947.11	323	161	27965.76	99.933%
	DRL w/ aggregation	300	**27952.63**	**353**	**233**	27965.76	99.953%
	Greedy	500	80779.23	25	0	81103.99	99.781%
	DRL w/o aggregation	500	81022.60	71	168	81103.99	99.899%
	DRL w/ aggregation	500	**81064.99**	**136**	**304**	81103.99	99.951%

very close to the optimal solutions and DRL with aggregation perform better than DRL without aggregation.

Comparison with Greedy and DRL without aggregation. The results show that the proposed DRL-based methods, with or without aggregation, always perform better than the greedy algorithm, in terms of the average solution quality ($\overline{Val}$), the number of optimally solved instances ($\#_{opt}$), and the number of instances with highest solution value ($\#_{highest}$). As shown in table 2, the state aggregation strategy improves the solutions of greedy algorithm for large instances, which is clearly observable in the solutions of all problem instance types. The column $\overline{Val}$ shows the difference between three approaches. The DRL with state aggregation always works better than DRL without aggregation and greedy algorithm. In terms of number of optimally solved instances, the DRL with aggregation greatly outperforms greedy algorithm (about ten times higher for large instances with 500 items). DRL with aggregation also works better than DRL without aggregation and solves

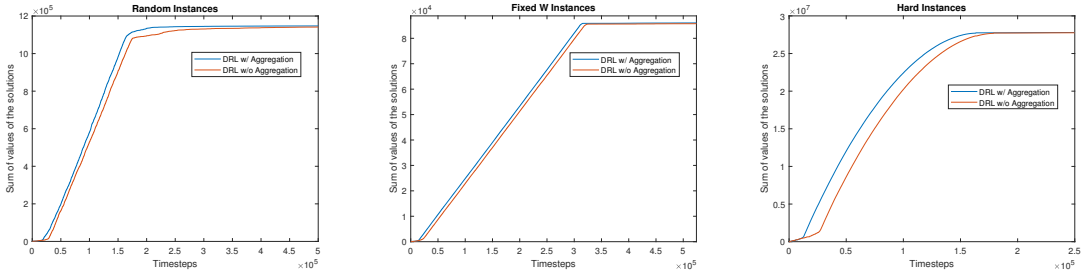


(a) RI, 300 items (b) FI, 300 items (c) HI, 300 items



(d) RI, 500 items (e) FI, 500 items (f) HI, 500 items

Figure 2: The box plot for the optimality gaps of solutions of the three instances types with 300 and 500 items.



(a) Random Instances (b) Fixed W_P instances (c) Hard instances

Figure 3: Learning rate of DRL algorithms w/ and w/o aggregation for 300instances

more instances optimally. Furthermore, based on $\#_{highest}$, the greedy algorithm never finds better solutions than DRL algorithms. Comparing to DRL without aggregation, the DRL with finds better solution most of the time (304 vs. 168 for HI and 500 items). Based on these results, DRL with aggregation method outperform the well-known greedy heuristic for KPs and also DRL without aggregation.

Learning Rate. The important benefit of DRL with aggregation method is that it is able to find the high quality solutions in less time steps. As it can be observed from Figures 3(a), 3(b) and 3(c), the learning rate of DRL with aggregation method is higher than the

DRL without aggregation. Hence, in general, it not only provides better solutions, but also the solutions are found in around $10,000$ fewer timesteps.

Comparison with (Bello et al., 2017). The pointer network based DRL method is used to solve KPs in (Bello et al., 2017). The FI instances with 50, 300 and 500 items and fixed capacities used in our experiments are generated in the same way as those used in (Bello et al., 2017). For these FI instances, our proposed DRL with aggregation method finds optimal solutions for 50 items, while a similar performance is reported in (Bello et al., 2017). Our method also finds close to optimal solutions for larger instances up to 500 items. In (Bello et al., 2017), the authors did not test instances with more than 200 items.

One disadvantage of the method of (Bello et al., 2017) is that it can only be applied to solve the instances with exactly same number of items N, and in addition, with exactly same capacity value W_P. Hence, it is not applicable for solving the instances of RI and HI that contain varying capacities and item numbers.

Comparison with (Gu and Hao, 2018). This work employs the pointer network and supervised learning to solve 0-1 KPs. The authors of (Gu and Hao, 2018) generated 10000 instances for training and 1000 for testing and selected 100 randomly selected test instance for evaluation. We generate the RI instances in the same way. Although our method is based on active searching, we followed a greedy algorithm according to the outputs of the policy network using 100 randomly selected instances that are not considered in active searching to achieve a fair comparison with (Gu and Hao, 2018). The results show that $\frac{\overline{Val}}{\overline{Val}_{opt}}$ ranges from 0.84 to 0.99 for the selected instances with 500 items. As a comparison, the performance ratio in (Gu and Hao, 2018) is 60% for a set of randomly generated problem instances with 500 items.

6. Conclusion and future work

We developed a DRL-based method for boosting the heuristic greedy algorithm and solving KP. In the DRL based KP solver, a policy DNN and a value DNN are trained using A2C algorithm and the policy DNN is used for sequentially selecting items to find a solution. The states in DRL modeling of KP contain the information of the instances that are aggregated to reduce the state space. The state aggregation policy is derived by solving a tabular RL problem. Using this aggregation policy, a state embedding is obtained and this state embedding is used with another RL framework to train the parameters of the policy network. We compared this method with several other approaches, namely the existing DRL approaches (i.e. Bello et al. (2017) and Gu and Hao (2018)), the greedy algorithm, and moreover, our approach without state aggregation, using three types of problem instances and with different problem sizes.

The results have demonstrated that the proposed approach is promising for solving the KP type of the problems which emerge in many applications such as cargo loading and capital budgeting. The proposed method with some adaptation effort may be generalized to some other COPs.

This paper uses RL to automate the reduction of the state space, as a pre-processing step of the DRL based approach for KPs. In general, automating the state, reward and action derivation for RL problems are interesting topics for research in the future.

References

Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *ICLR (Workshop)*, 2017. URL `https://academic.microsoft.com/paper/2560592986`.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*, 2018.

Stephen Cook. The p versus np problem. *The millennium prize problems*, pages 87–104, 2006.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711, 2016.

Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2008.

Ding-Zhu Du and Panos M Pardalos. *Handbook of combinatorial optimization: supplement*, volume 1. Springer Science & Business Media, 2013.

Yanhong Feng, Juan Yang, Congcong Wu, Mei Lu, and Xiang-Jun Zhao. Solving 0–1 knapsack problems by chaotic monarch butterfly optimization algorithm with gaussian mutation. *Memetic Computing*, 10(2):135–150, 2018.

Shenshen Gu and Tao Hao. A pointer network based deep learning algorithm for 0–1 knapsack problem. In *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, pages 473–477. IEEE, 2018.

LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL `http://www.gurobi.com`.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

Tingfei Huang, Yang Ma, Yuzhen Zhou, Honglan Huang, Dongmei Chen, Zidan Gong, and Yao Liu. A review of combinatorial optimization with graph neural networks. In *2019 5th International Conference on Big Data and Information Analytics (BigDIA)*, pages 72–77. IEEE, 2019.

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

Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *ICLR 2019 : 7th International Conference on Learning Representations*, 2019.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, 2005.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.

Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

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

Christophe Wilbaut, Said Hanafi, and Said Salhi. A survey of effective heuristics and their application to a variety of knapsack problems. *IMA Journal of Management Mathematics*, 19(3):227–244, 2008.

Gerhard J Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial optimization—eureka, you shrink!*, pages 185–207. Springer, 2003.

Nitin Yadav, Carsten Murawski, Sebastian Sardina, and Peter Bossaerts. Phase transition in the knapsack problem. *arXiv preprint arXiv:1806.10244*, 2018.