



温州肯恩大学

WENZHOU-KEAN UNIVERSITY

# A Comparative Study of Dynamic Programming, Graph Neural Networks, and Reinforcement Learning for the 0–1 Knapsack Problem

CPS 3440: Analysis of Algorithms

Supervisor: Omar Dib

December 15, 2025

<b>Guanlin Li</b> (Team Leader)	1308245 (W07)
Chunguang Lu	1365419 (W08)
Xiaoqian Zhang	1365436 (W08)
Mingshi Cai	1365432 (W08)
Yiyue Yin	1235600 (W07)

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Problem Background . . . . .	2
1.2	Research Motivation . . . . .	2
<b>2</b>	<b>Methods</b>	<b>3</b>
2.1	Problem Formulation (0–1 Knapsack) . . . . .	3
2.2	Method 1: Dynamic Programming (DP) . . . . .	3
2.3	Method 2: Graph Neural Network (GNN) Approach . . . . .	4
2.3.1	Graph Construction . . . . .	4
2.3.2	Node Embedding . . . . .	4
2.3.3	Message Passing . . . . .	4
2.3.4	Readout and Decision Making . . . . .	5
2.3.5	Training Pipeline . . . . .	5
2.3.6	Feasible Solution Decoding . . . . .	5
2.3.7	Evaluation Metrics . . . . .	6
2.4	Method 3: Reinforcement Learning with Deep Q-Network . . . . .	6
2.4.1	Problem Formulation . . . . .	6
2.4.2	State, Action, and Reward Definition . . . . .	6
2.4.3	Deep Q-Network Architecture . . . . .	7
2.4.4	Training Components . . . . .	7
2.4.5	Inference Phase . . . . .	7
2.4.6	Modeling Considerations . . . . .	8
2.4.7	Algorithm Summary . . . . .	8
<b>3</b>	<b>Experimental Setup and Results</b>	<b>8</b>
3.1	Dataset . . . . .	8
3.2	Methods Compared (DP, GNN, DQN) . . . . .	9
3.3	Results . . . . .	9
<b>4</b>	<b>Analysis and Conclusion</b>	<b>11</b>
<b>5</b>	<b>Future Work</b>	<b>11</b>

# 1 Introduction

## 1.1 Problem Background

Combinatorial optimization plays a central role in many real-world decision-making problems, such as logistics scheduling, portfolio optimization, resource allocation, and network design. Many of these problems are NP-hard, which makes exhaustive search infeasible even for instances of moderate size. Among them, the *0–1 knapsack problem* serves as a canonical benchmark: given a set of items, each associated with a weight and a value, and a knapsack with limited capacity, the objective is to select a subset of items—where each item can be either selected or not—to maximize the total value without exceeding the capacity constraint.

Traditionally, the 0–1 knapsack problem can be solved exactly using *Dynamic Programming (DP)*. DP exploits the structure of the problem to compute optimal solutions in pseudo-polynomial time and therefore provides a reliable ground-truth baseline for small to medium-sized instances. However, as the number of items or the capacity grows, DP may become increasingly expensive in terms of runtime and memory consumption, especially in scenarios where decisions must be made repeatedly or under real-time constraints.

In recent years, learning-based approaches have attracted growing attention as alternatives for solving combinatorial optimization problems. Models such as *Graph Neural Networks (GNNs)* and other deep learning methods aim to learn a direct mapping from problem instances to high-quality solutions through offline training, enabling fast inference at test time. In parallel, *Reinforcement Learning (RL)* formulates the knapsack problem as a sequential decision-making process, in which an agent incrementally selects items by interacting with an environment and receiving rewards, typically under the *Markov Decision Process (MDP)* framework. These approaches offer new perspectives on scalability and generalization, but their effectiveness relative to classical exact methods remains an important empirical question.

## 1.2 Research Motivation

Although Dynamic Programming can produce exact solutions for the 0–1 knapsack problem, its computational cost typically scales as  $O(nW)$ , where  $n$  denotes the number of items and  $W$  the knapsack capacity. When either parameter becomes large, DP quickly encounters practical limitations in both runtime and memory usage. Moreover, DP generally needs to be executed from scratch for each new instance, which makes it less suitable for online or high-throughput settings where many instances must be solved efficiently.

In contrast, learning-based methods treat knapsack instances as data and aim to learn a reusable decision policy through offline training. Once trained, such models can perform inference with significantly lower per-instance computational cost and may generalize to previously unseen instances. Among these approaches, *Graph Neural Networks (GNNs)* are particularly appealing due to their ability to model structured inputs: items can naturally be represented as nodes, while message passing allows the model to capture interactions and dependencies that go beyond simple hand-crafted heuristics. *Reinforcement Learning*, on the other hand, provides a more generic formulation by constructing solutions sequentially, but may suffer from instability and high variance during training.

Motivated by these considerations, this report conducts a *systematic comparison* of three representative approaches—*Dynamic Programming (DP)* as an exact classical baseline, *Graph Neural Networks (GNNs)* as a structure-aware learning-based method, and *Reinforcement Learning (RL/DQN)* as a general sequential decision-making approach—under a *unified dataset and evaluation protocol*. The comparison focuses on solution quality, computational efficiency, scalability, and generalization behavior in the context of the 0–1 knapsack problem.

Specifically, this project aims to address the following research questions:

- How closely can learning-based methods approximate DP-optimal solutions under identical data and evaluation settings?
- What are the trade-offs between solution quality and runtime efficiency among DP, GNN, and RL-based approaches?
- How do different modeling paradigms—global reasoning (GNN) versus sequential decision-making (RL)—affect performance on the 0–1 knapsack problem?

## 2 Methods

### 2.1 Problem Formulation (0–1 Knapsack)

We first consider the classic 0–1 knapsack problem. Let there be  $n$  items indexed by  $i = 1, \dots, n$ . Each item  $i$  is associated with:

- a weight  $w_i > 0$ ,
- a value  $v_i > 0$ .

The knapsack has a maximum capacity  $W > 0$ . For each item  $i$ , we introduce a binary decision variable  $x_i \in \{0, 1\}$ , where  $x_i = 1$  indicates that item  $i$  is selected, and  $x_i = 0$  otherwise.

The goal is to select a subset of items such that the total weight does not exceed the capacity, while the total value is maximized. The 0–1 knapsack problem can be formulated as the following integer program:

$$\max \sum_{i=1}^n v_i x_i \quad \text{subject to} \quad \sum_{i=1}^n w_i x_i \leq W, \quad x_i \in \{0, 1\}.$$

We denote by  $\mathcal{S}^*$  an optimal solution and by  $v^*$  the corresponding optimal objective value. This formulation serves as the basis for our dynamic programming approach, as well as a reference for supervised or reinforcement learning targets in our learning-based methods.

### 2.2 Method 1: Dynamic Programming (DP)

Dynamic Programming (DP) provides one of the most classical and widely used exact algorithms for solving knapsack problems. Due to the structure of the knapsack constraint—which aggregates item contributions linearly—DP excels at exploring feasible combinations in a principled manner. Although the knapsack problem is NP-hard in general, DP achieves pseudo-polynomial time by exploiting the numerical value of the capacity  $W$ . In this section, we review the standard DP formulation for the 0–1 knapsack problem.

In the 0–1 knapsack problem, each item  $i \in \{1, \dots, n\}$  can either be selected or not selected. Let  $w_i$  and  $v_i$  denote the weight and value of item  $i$ , respectively, and let  $W$  be the knapsack capacity. A standard DP state is defined as the maximum total value achievable using the first  $i$  items with capacity  $w$ .

This yields a time complexity of  $O(nW)$  and a space complexity of  $O(nW)$ . In practice, the DP table can be compressed to a one-dimensional array  $dp[c]$  using  $O(W)$  memory, where capacities must be iterated in descending order to prevent reusing the same item multiple times.

#### Core Code

```

1  for i in range(1, n + 1):
2      wi = weights[i - 1]
3      vi = values[i - 1]
4      for w in range(capacity + 1):
5          dp[i][w] = dp[i - 1][w]
6          if wi <= w:
7              dp[i][w] = max(dp[i][w], dp[i - 1][w - wi] + vi)

```

Listing 1: Dynamic Programming Update Rule for the 0–1 Knapsack Problem

As shown in Listing 1, the DP algorithm updates the optimal value by either excluding or including the current item when feasible.

## 2.3 Method 2: Graph Neural Network (GNN) Approach

The 0–1 knapsack problem is a classic combinatorial optimization task, where items must be selected under a capacity constraint to maximize total value. While Dynamic Programming (DP) can compute exact optimal solutions and therefore serves as a ground-truth oracle, its computational cost grows quickly as the instance size increases.

Graph Neural Networks (GNNs) provide a learning-based alternative by modeling each knapsack instance as a graph and learning to reason over item-level features and their interactions through message passing. After offline training, GNN inference is typically much faster than DP and can produce high-quality approximate solutions.

### 2.3.1 Graph Construction

GNNs are well suited for problems with an inherent *entity–relationship* structure. In the knapsack problem, each item can be treated as an entity, while relationships between items can be defined based on feature similarity or global constraints.

For each knapsack instance, we construct a graph  $G = (V, E)$ , where each node represents an item.

**Nodes** In the 0–1 knapsack setting, each node corresponds to an item  $i$ . Node features may include:

- item weight  $w_i$ ,
- item value  $v_i$ ,
- value density  $v_i/(w_i + \varepsilon)$ ,
- normalized features such as  $w_i/W$  and  $v_i/\max(v)$ .

**Edges** The knapsack problem does not naturally define explicit edges between items. Therefore, we consider common graph construction strategies:

- **Fully connected graph**, which connects all pairs of items and provides strong expressivity at a higher computational cost;
- **$k$ -nearest neighbor (kNN) graph**, where each node is connected to its  $k$  most similar neighbors in feature space, offering a more efficient approximation.

Optionally, a global capacity node can be added and connected to all item nodes to explicitly inject capacity information into message passing.

### 2.3.2 Node Embedding

Node embedding maps raw item features into a latent vector space suitable for learning. An initial embedding is typically obtained through a linear transformation:

$$h_i^{(0)} = \text{Linear}(x_i),$$

where  $x_i$  denotes the feature vector of item  $i$ .

### 2.3.3 Message Passing

Message passing is the core mechanism of GNNs. At each layer, nodes aggregate information from their neighbors and update their own representations. A typical message-passing layer consists of the following steps:

1. Neighbor message computation,

2. Message aggregation (e.g., sum or mean),
3. Representation update using the aggregated message and the node’s current state.

Formally, the embedding of node  $i$  at layer  $l$  can be expressed as:

$$h_i^{(l)} = \phi \left( h_i^{(l-1)}, \text{AGG} \left( \{h_j^{(l-1)} \mid j \in \mathcal{N}(i)\} \right) \right),$$

where  $\phi(\cdot)$  denotes a learnable update function and  $\mathcal{N}(i)$  is the set of neighbors of node  $i$ .

Stacking multiple GNN layers allows information to propagate across multiple hops, enabling each node to capture both local and global context.

### 2.3.4 Readout and Decision Making

After message passing, node embeddings are mapped to selection probabilities using a multi-layer perceptron (MLP):

$$p_i = \sigma(Wh_i + b),$$

where  $\sigma(\cdot)$  is the sigmoid function.

Final binary decisions are obtained by thresholding:

$$x_i = \begin{cases} 1, & p_i > 0.5, \\ 0, & \text{otherwise.} \end{cases}$$

### 2.3.5 Training Pipeline

We adopt a supervised training strategy using DP-optimal solutions as labels. The loss function consists of two components:

- **Classification loss:** Binary Cross-Entropy (BCE) between predicted probabilities and DP labels;
- **Capacity constraint penalty:**

$$\lambda \cdot \max \left( 0, \sum_i w_i x_i - W \right),$$

which discourages infeasible solutions.

The total loss is given by:

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{penalty}}.$$

### 2.3.6 Feasible Solution Decoding

Raw GNN outputs do not automatically satisfy the capacity constraint. Therefore, we apply a decoding procedure to ensure feasibility. A common strategy is greedy feasible decoding: items are sorted by predicted scores, and selected sequentially while respecting the capacity constraint. This guarantees that  $\sum_i w_i x_i \leq W$ , enabling a fair comparison with DP.

#### Core Code

```

1 # h: node embeddings after message passing
2 scores = mlp(h) # selection scores
3 probs = sigmoid(scores) # selection probabilities
4
5 # greedy feasible decoding
6 selected = []
7 remaining_capacity = W
8 for i in sorted_items_by_score(probs):
9     if weights[i] <= remaining_capacity:
10         selected.append(i)
11         remaining_capacity -= weights[i]
```

Listing 2: GNN Inference and Decision Pipeline for 0–1 Knapsack

### 2.3.7 Evaluation Metrics

To compare GNN with DP (and other methods), we report:

- optimality gap relative to DP,
- feasibility rate,
- runtime per instance,
- generalization performance under varying problem sizes.

## 2.4 Method 3: Reinforcement Learning with Deep Q-Network

### 2.4.1 Problem Formulation

We formulate the 0–1 knapsack problem as a sequential decision-making task and solve it using a Deep Q-Network (DQN). At each step  $t$ , the agent decides whether to select or skip the current item  $i_t$ , and the episode proceeds sequentially until all  $n$  items have been considered. This process naturally fits the Markov Decision Process (MDP) framework, which is defined as follows.

**State ( $S$ ).** The state is represented as a tuple  $(i, c)$ , where  $i \in \{0, \dots, n\}$  denotes the index of the current item under consideration, and  $c \in \{0, \dots, W\}$  denotes the remaining capacity of the knapsack. The episode terminates when  $i = n$ .

**Action ( $A$ ).** At each state, the agent selects a binary action:

- $a = 0$ : skip the current item;
- $a = 1$ : select the current item, provided that the remaining capacity allows it.

**Transition.** The environment dynamics are deterministic. If the agent skips the item ( $a = 0$ ), the next state is  $(i + 1, c)$ . If the agent selects the item ( $a = 1$ ) and its weight  $w_i \leq c$ , the next state becomes  $(i + 1, c - w_i)$ . If  $a = 1$  but  $w_i > c$ , the action is considered invalid and is either masked or penalized.

**Reward ( $R$ ).** The reward function is defined to encourage valid and valuable selections:

- A positive reward equal to the item value  $v_i$  is given when an item is selected and fits within the remaining capacity;
- A reward of zero is assigned when the item is skipped;
- A negative penalty is applied when an invalid action is taken.

Optionally, the total accumulated value can be deferred and provided only at the terminal state.

**Policy and Learning.** The agent follows an  $\varepsilon$ -greedy policy to balance exploration and exploitation, with  $\varepsilon$  decaying over time. Episodes start from the initial state  $(0, W)$  and are repeated for a large number of iterations. The Q-values are updated according to the standard Q-learning rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right),$$

where  $\alpha$  is the learning rate and  $\gamma$  is the discount factor.

### 2.4.2 State, Action, and Reward Definition

**State Representation** The state  $s_t$  at step  $t$  is represented by a fixed-length feature vector encoding:

- Current item index  $t$

- Remaining capacity  $c_t$
- Summary statistics of remaining items (e.g., mean value, mean weight)

Formally:

$$s_t = [t, c_t, \bar{v}_t, \bar{w}_t] \in \mathbb{R}^4$$

where  $\bar{v}_t$  and  $\bar{w}_t$  denote the average value and weight of items from  $t$  to  $n$ .

**Action Space** The action space is binary:

$$\mathcal{A} = \{0, 1\}$$

where:

- $a_t = 1$ : select the current item
- $a_t = 0$ : skip the current item

**Reward Function** The reward  $r_t$  is defined as:

$$r_t(s_t, a_t) = \begin{cases} v_t & \text{if } a_t = 1 \text{ and } w_t \leq c_t \\ 0 & \text{if } a_t = 0 \\ -\rho & \text{if } a_t = 1 \text{ and } w_t > c_t \quad (\text{infeasibility penalty}) \end{cases}$$

where  $v_t$  and  $w_t$  are the value and weight of item  $t$ ,  $c_t$  is the remaining capacity, and  $\rho > 0$  is a penalty parameter.

### 2.4.3 Deep Q-Network Architecture

The Q-function  $Q(s, a; \theta)$  is approximated by a multi-layer perceptron (MLP) with parameters  $\theta$ :

$$Q(s, a; \theta) = f_{\text{MLP}}(s; \theta)$$

The network outputs Q-values for both actions:  $[Q(s, 0; \theta), Q(s, 1; \theta)]$ .

### 2.4.4 Training Components

- **Experience Replay Buffer:** Stores transitions  $(s_t, a_t, r_t, s_{t+1})$  for stable learning
- **Target Network:** Separate network  $Q(s, a; \theta^-)$  with periodic updates
- **Loss Function:** Mean squared Bellman error

$$\mathcal{L}(\theta) = \mathbb{E}_{(s, a, r, s') \sim \mathcal{D}} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right]$$

where  $\gamma \in [0, 1]$  is the discount factor and  $\mathcal{D}$  is the replay buffer

- **Exploration Strategy:**  $\epsilon$ -greedy with decay

### 2.4.5 Inference Phase

After training, the DQN policy  $\pi$  is evaluated greedily without exploration:

$$\pi(s_t) = \arg \max_{a \in \mathcal{A}_{\text{feasible}}(s_t)} Q(s_t, a; \theta^*)$$

where  $\mathcal{A}_{\text{feasible}}(s_t) \subseteq \mathcal{A}$  contains only actions that do not violate the capacity constraint. The episode terminates after processing all  $n$  items, yielding a valid solution  $\mathcal{S} \subseteq \{1, \dots, n\}$ .



### 2.4.6 Modeling Considerations

Based on the MDP formulation described above, a tabular Q-learning approach was initially considered as a baseline. However, it was found to be impractical for the problem scale addressed in this work due to several fundamental limitations.

First, the size of the state space grows rapidly with the number of items and the knapsack capacity. Since the state is defined as  $(i, c)$ , the total number of states satisfies  $|\mathcal{S}| = O(n \cdot W)$ , which becomes prohibitively large as either  $n$  or  $W$  increases.

Second, additional state statistics used to improve decision quality, such as the average remaining item value  $\bar{v}_t$  and average remaining item weight  $\bar{w}_t$ , are continuous-valued. This further complicates tabular representations and discretization, leading to increased memory usage and unstable learning behavior.

Third, tabular Q-learning lacks generalization across problem instances. Each new item set or capacity configuration would require retraining a separate Q-table, resulting in significant computational overhead and poor scalability.

To address these challenges, a Deep Q-Network (DQN) is adopted, which replaces the explicit Q-table with a neural network function approximator. By learning a mapping from states to action values, the DQN enables generalization across different knapsack instances and allows a single trained model to handle varying item sets and capacity constraints. As a result, DQN provides a scalable and flexible reinforcement learning solution suitable for the experimental setting considered in this study.

### 2.4.7 Algorithm Summary

---

#### Algorithm 1 DQN for 0–1 Knapsack Problem

---

**Require:** Item set  $\{(v_i, w_i)\}_{i=1}^n$ , capacity  $W$

**Ensure:** Trained Q-network  $Q_{\theta^*}$

---

```

1: Initialize Q-network  $Q_{\theta}$ , target network  $Q_{\theta^-}$ , replay buffer  $\mathcal{D}$ 
2: for episode = 1 to  $M$  do
3:   Initialize state  $s_0$ , remaining capacity  $c_0 = W$ 
4:   for  $t = 0$  to  $n - 1$  do
5:     Choose action  $a_t$  using  $\epsilon$ -greedy policy
6:     Execute  $a_t$ , observe reward  $r_t$  and next state  $s_{t+1}$ 
7:     Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{D}$ 
8:     Sample random mini-batch from  $\mathcal{D}$ 
9:     Compute target:  $y = r + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-)$ 
10:    Update  $\theta$  using gradient descent on loss  $(y - Q(s_t, a_t; \theta))^2$ 
11:    Every  $C$  steps: update  $\theta^- \leftarrow \theta$ 
12:   end for
13: end for
14: return  $Q_{\theta^*}$ 

```

---

## 3 Experimental Setup and Results

### 3.1 Dataset

To ensure a fair comparison across methods, we evaluate DP, GNN, and DQN on the same fixed dataset of 0–1 knapsack instances. The dataset contains 1000 medium-difficulty instances stored as `.npz` files under `dataset/knapsack01_medium/`. Each file corresponds to one instance and includes: `weights` (1D array), `values` (1D array), and `capacity` (scalar).

The dataset is treated as read-only throughout the entire pipeline: no script modifies the `.npz` files on disk. Any required compatibility handling (e.g., tensor conversion or normalization) is performed in memory during loading. This fixed dataset is shared by all methods without modification, ensuring a fair and consistent evaluation across DP, GNN, and DQN.

In our implementation, the dataset is generated offline using our instance generation script (e.g., `data_generate_01.py`), and then reused identically by the DP, GNN, and DQN training/evaluation programs.

### 3.2 Methods Compared (DP, GNN, DQN)

**Dynamic Programming (DP).** At the early stage of the project, Dynamic Programming (DP) was treated as one of the candidate methods and evaluated alongside learning-based approaches. Through preliminary experiments, we observed that DP consistently produced optimal solutions for all tested instances. As a result, DP was subsequently designated as the ground-truth oracle and used as the reference baseline for evaluating solution quality.

In the final experimental setup, we use a classic 0–1 knapsack DP solver purely as an evaluation oracle. The baseline script (e.g., `Dynamic_Programming/dp_baseline_eval.py`) runs DP on all 1000 instances, computes the optimal objective value, and backtracks the selected items. The outputs are saved to `results/DP/dp_results.csv`, which serves as the reference for all optimality-gap calculations. DP does not involve any learning process.

**Graph Neural Network (GNN).** The GNN approach is trained in a supervised manner using DP-optimal solutions as labels. After offline training, inference consists of a single forward pass followed by greedy feasibility decoding to ensure that the predicted solution satisfies the capacity constraint. Training is conducted via `Graph_Neural_Network/run_train.py`, and the trained model is saved to `results/GNN/gnn.pt`. Evaluation is performed by `Graph_Neural_Network/evaluate.py`, producing `results/GNN/gnn_eval_results.csv`. The output format is aligned with the DP and DQN results to enable consistent comparison.

**Reinforcement Learning (DQN).** In the reinforcement learning baseline, we initially explored tabular Q-learning. However, due to the rapidly growing state space and difficulties in unifying dataset-based training, tabular Q-learning was found to be impractical for our experimental setting. Consequently, we adopted a Deep Q-Network (DQN) formulation, which replaces the Q-table with a neural network function approximator.

The final RL baseline uses DQN with a replay buffer and a target network. Training is performed by `Reinforcement_Learning/dqn_knapsack_project/train_dqn.py`, which scans the fixed dataset directory and saves the trained policy to `results/DQN/dqn.pt`. For evaluation, `Reinforcement_Learning/dqn_knapsack_project/evaluate_dqn.py` runs greedy inference (i.e., without exploration) on all 1000 instances and outputs `results/DQN/eval_results.csv`.

**Unified evaluation and result merging.** All methods are evaluated on the same dataset with identical per-instance settings. The resulting outputs are merged using `tools/merge_results.py` into a unified comparison table, which is then used to compute aggregate statistics such as average runtime and optimality gap.

### 3.3 Results

Table 1 reports the average runtime, optimality gap, and accuracy of the three methods on 1000 0–1 knapsack instances, with Dynamic Programming (DP) serving as the ground-truth oracle. By definition, DP achieves zero gap and 100% accuracy, confirming its role as the exact reference baseline.

The Graph Neural Network (GNN) achieves an average runtime of 2.14 ms, which is comparable to the 2.21 ms of DP. At the same time, GNN maintains a small average optimality gap of 0.0168, corresponding to an accuracy rate of 98.32%. This result indicates that the GNN is able to approximate DP-level solutions with minimal loss in solution quality while retaining very low inference cost.

In contrast, the Deep Q-Network (DQN) exhibits a substantially higher average runtime of 19.72 ms and a larger average optimality gap of 0.1516, resulting in an accuracy rate of 84.84%. These results suggest that, under the current experimental setup, the RL-based approach incurs higher computational cost and produces solutions that are further from the optimal baseline.

Beyond average performance, Table 2 and Table 3 provide additional insight into solution stability. While GNN demonstrates relatively low variance across instances, DQN exhibits considerably higher variability, with some instances achieving near-optimal solutions and others deviating significantly. This higher variance partially explains the larger average gap observed for DQN and highlights the challenge of stable sequential decision-making in the knapsack setting.

It is worth noting that the DQN model was trained under a limited computational budget. Due to hardware constraints, the total number of training steps was reduced from the initially planned 200k to 50k steps. This design choice was made to maintain a reasonable and fair comparison with GNN training cost within the same experimental environment. The observed performance gap of DQN should therefore be interpreted in the context of this reduced training budget.

Table 1: Average performance comparison on 1000 0–1 knapsack instances. DP is used as the ground-truth oracle. *The DQN model was trained for 50k steps (reduced from the originally planned 200k steps due to hardware constraints).*

Method	Avg. Time (ms)	Avg. Gap	Accuracy Rate
DP	2.21	0.0000	100.00%
GNN	2.14	0.0168	98.32%
DQN	19.72	0.1516	84.84%

Table 2: Performance comparison with stability statistics (mean  $\pm$  standard deviation). Lower gap and lower variance indicate better and more stable performance.

Method	Time (ms)	Gap	Accuracy (%)
DP	$2.21 \pm 1.18$	$0.0000 \pm 0.0000$	$100.00 \pm 0.00$
GNN	$2.14 \pm 0.36$	$0.0168 \pm 0.0097$	$98.32 \pm 0.97$
DQN	$19.72 \pm 5.21$	$0.1516 \pm 0.0404$	$84.84 \pm 4.04$

Table 3: Distribution statistics of optimality gap (lower is better). IQR denotes the interquartile range (Q1–Q3), and P95 denotes the 95th percentile.

Method	Median Gap	IQR (Q1–Q3)	P95 Gap	Gap $\leq$ 1%
GNN	0.0152	0.0101–0.0220	0.0336	24.7%
DQN	0.1509	0.1244–0.1786	0.2167	0.0%

## 4 Analysis and Conclusion

In this study, we conducted a systematic comparison of three representative approaches for solving the 0–1 knapsack problem: Dynamic Programming (DP), a supervised Graph Neural Network (GNN), and a Reinforcement Learning method based on Deep Q-Networks (DQN). All methods were evaluated under a unified experimental pipeline using the same dataset, evaluation protocol, and optimality-gap-based metrics.

Dynamic Programming consistently achieved zero optimality gap and 100% accuracy, confirming that it provides exact optimal solutions and is well suited as a ground-truth oracle. Although DP does not yield the fastest runtime in this setting, its primary value lies in serving as a reliable accuracy benchmark rather than in inference speed. Due to its pseudo-polynomial time and space complexity with respect to the capacity, DP is more appropriate for offline evaluation than for large-scale or real-time deployment.

The Graph Neural Network demonstrates the best overall trade-off between efficiency and solution quality. It achieves an average runtime comparable to DP while maintaining a small optimality gap and high accuracy. Moreover, stability analysis shows that GNN exhibits relatively low variance across instances, indicating that it can consistently approximate DP-level solutions. These results suggest that GNNs are effective at capturing the structural patterns of the knapsack problem and can amortize computation through offline training while retaining fast inference.

In contrast, the DQN-based reinforcement learning approach exhibits both higher runtime and lower solution quality under the current setup. In addition to a larger average gap, DQN shows substantially higher variance across instances, with some solutions close to optimal and others deviating significantly. This instability is primarily attributed to the sequential nature of decision-making in RL and the sensitivity of performance to state representation and training budget. Due to hardware constraints, the DQN model was trained for a reduced number of steps (50k instead of the originally planned 200k), which was chosen to maintain a reasonable and fair comparison with the training cost of the GNN. The observed performance of DQN should therefore be interpreted in the context of this limited computational budget.

Overall, our results indicate that DP serves as an exact but computationally constrained baseline, GNN provides a strong balance between speed, accuracy, and stability, and DQN highlights both the potential and challenges of sequential reinforcement learning for combinatorial optimization. Future work may explore improved RL architectures, richer state representations, or increased training budgets to reduce variance and improve solution quality.

## 5 Future Work

An interesting direction for future work is to explore hybrid approaches that combine the strengths of Graph Neural Networks and Reinforcement Learning. In our experiments, GNN demonstrates strong global reasoning ability and stable performance, while the RL approach provides a flexible framework for sequential decision-making. One possible extension is to use GNNs to learn compact, structure-aware state representations, which can then be integrated into an RL framework to guide action selection. Such a hybrid design may help reduce variance in RL training while preserving its ability to handle sequential constraints. We leave this direction for future investigation.