E-Companion – "Learning Virtual Machine Scheduling in Cloud Computing through Language Agents"

Jiehao Wu[#], Ziwei Wang[#], Junjie Sheng, Wenhao Li, Xiangfeng Wang[*], Jun Luo[*]
jhwu@stu.ecnu.edu.cn, zoey_wangzw@sjtu.edu.cn, jarvis@stu.ecnu.edu.cn, whli@tongji.edu.cn,
xfwang@cs.ecnu.edu.cn, jluo_ms@sjtu.edu.cn

EC.1. Detailed Literature Review

The Dynamic Multidimensional Bin Packing (DMBP) problem is a complex extension of the classical bin packing problem, characterized by the sequential allocation of multidimensional items that arrive and depart dynamically. This problem requires the efficient scheduling of shared resources such as CPU, memory, and bandwidth. Unlike scenarios involving single-unit capacity occupation, DMBP necessitates intricate multidimensional resource sharing, making it a computationally intractable NP-hard combinatorial optimization problem. To address this, researchers employ optimization-based techniques, learning-based methods, or heuristics to derive approximate solutions (Coffman et al. 1984).

Optimization-based Algorithms for DMBP. Offline DMBP assumes that all items to be packed and their sequence are known in advance. It can be formulated as a mixed-integer programming problem, which can be solved exactly using branch-and-bound algorithms (Martello et al. 2000). These algorithms are embedded in commercial solvers such as Gurobi (Gurobi Optimization, LLC 2024). For large-scale problems, decomposition methods have proven effective in solving DMBP (Pisinger and Sigurd 2007, Puchinger and Raidl 2007, Côté et al. 2021). In the online setting, only the current item information is known, and items must be packed sequentially in an unknown arrival process. When item sizes are stochastic, the problem becomes even more challenging, and most theoretical research focuses on analyzing the multidimensional and dynamic characteristics separately (Chen et al. 2023). For a review of approximation methods for online multidimensional bin packing problems, refer to Christensen et al. (2017). In contrast, much of the literature on online one-dimensional dynamic bin packing focuses on applications such as surgery scheduling, where emergency patients arrive randomly and require immediate service with uncertain operation durations (Berg and Denton 2017, Zhang et al. 2020). From an OM perspective, dynamic bin packing can be viewed as a service system, as discussed by Maguluri et al. (2012), Stolyar (2013), and Stolyar and Zhong (2021). For instance, Stolyar (2013) develops an infinite-server system model with homogeneous servers, allowing arriving VMs to be immediately assigned to a server. A series of studies by Stolyar (2013), Stolyar and Zhong (2021) reveal that simple randomized policies could achieve asymptotic optimality as the system scale, measured by the VM arrival rate and total number of customers, increases to infinity.

Learning-based Algorithms for DMBP. With the rapid development of machine learning (ML), researchers have explored its application to DMBP. One approach is to apply ML directly to solve DMBP. For instance, Zhao et al. (2021) combine deep learning and reinforcement learning (RL) techniques to handle

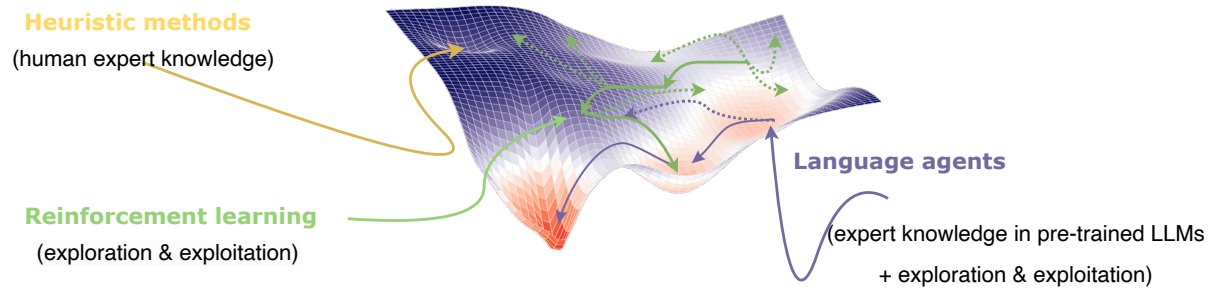
the dynamic nature and physical constraints of online packing, improving space utilization and operational efficiency. Another approach is to integrate RL with heuristics or mathematical models. RL optimizes the decision-making process for bin packing, where heuristic methods provide initial solutions or baselines. The RL algorithms then learn from these baselines and explore better solutions due to their exploratory nature (Zhang et al. 2022). Additionally, heuristic searches can provide supplementary reward signals to the RL agent, thereby accelerating the training process. Cai et al. (2019) propose an approach where the RL agent creates an initial feasible solution, which is subsequently optimized further using simulated annealing. Jiang et al. (2021) enhance solution quality by integrating RL with constraint programming (CP) by treating orientation and position as decision variables, in which the RL agent dynamically selects their values during a branch-and-bound search, finally improving solution efficiency within the CP framework.

Heuristic Algorithms for DMBP. To efficiently address the online DMBP problem, heuristic algorithms often derive insights from theoretical analysis to accelerate the search process, solving instances with tens of thousands of items within a limited time (Coffman et al. 1983, Hadary et al. 2020). Common algorithms include First-Fit (allocating the next item to the first bin with sufficient capacity) and Best-Fit (allocating the item to the bin with the smallest remaining capacity that still fits) (Garey et al. 1976). These algorithms provide performance bounds, making them viable as approximation methods. Notably, Azar et al. (2013) and Azar and Vainstein (2019) establish tight bounds for online multidimensional and dynamic DMBP, respectively. When uncertainty or nonstationarity is involved, Bentley et al. (1984) analyze First-Fit and Best-Fit algorithms for online bin packing with stochastic item sizes. Coffman and Stolyar (2001) study these algorithms in stochastic process settings, focusing on item arrivals governed by stochastic processes.

LLMs for Modeling and Code Generation. The integration of LLMs into combinatorial optimization has advanced two primary methodologies: automated mathematical modeling and heuristic algorithm generation. A critical challenge in optimization lies in transforming textual problem descriptions into formal mathematical formulations. Pioneering work in this domain includes the NL4OPT competition (Ramamonjison et al. 2023), which establishes a benchmark for translating natural language into mathematical programs using pre-trained language models. Building on this foundation, OptiMUS (AhmadiTeshnizi et al. 2024) introduce a modular framework to formulate and solve mixed-integer linear programming (MILP) problems from natural language inputs, demonstrating end-to-end capabilities from problem interpretation to solver code generation. Further extending the scope, Huang et al. (2025) propose a synthetic data generation framework to train LLMs across different optimization problem types, effectively addressing data scarcity challenges in specialized domains.

Beyond mathematical modeling, LLMs exhibit remarkable potential in designing heuristic algorithms for combinatorial optimization problems. For instance, Romera-Paredes et al. (2024) and Ye et al. (2024) leverage LLMs to iteratively generate and refine heuristic rules, outperforming traditional evolutionary algorithms in generalization capability and interpretability for problems like bin packing. Notably, these methods exploit

Figure EC.1 Method Comparison of Heuristic, Reinforcement Learning, and Language Agents Methods.



the code generation proficiency of LLMs to evolve context-aware strategies through systematic prompt engineering and fitness-guided evolutionary iterations.

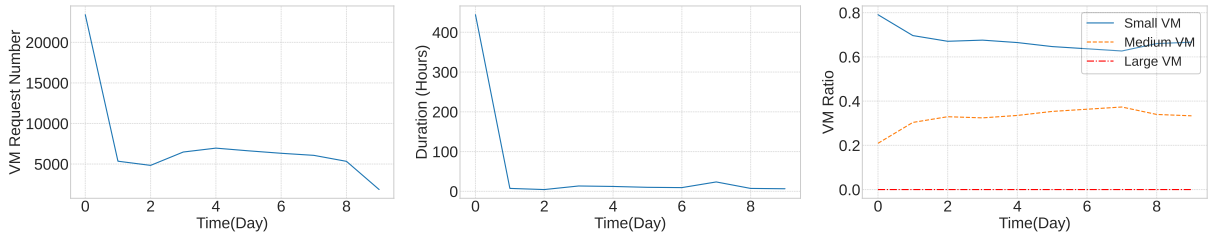
Limitations of Existing Methods and the Role of LLMs. Optimization-based approaches provide theoretical guarantees but face significant computational scalability challenges when addressing large-scale problems with multidimensional and dynamic characteristics. Learning-based methods, particularly RL, offer adaptive decision-making capabilities but typically require substantial computational resources for training. Traditional heuristic methods, while computationally efficient, are susceptible to local optima and heavily rely on manually designed rules based on domain expert knowledge. To address these limitations, we propose integrating the generalized expert knowledge embedded in LLMs with the exploration-exploitation mechanisms of RL frameworks. As illustrated in Figure EC.1, this approach aims to automate the design of hyper-heuristic algorithms, reducing reliance on human experts while adapting to dynamic environments. Our work applies this paradigm to Virtual Machine (VM) scheduling, transitioning from synthetic benchmarks to real-world industrial scenarios with dynamic resource demands and heterogeneous hardware constraints.

EC.2. Supplementary Experiments

We conduct synchronous experimental validation on the Azure dataset. The dataset *AzurePublicDatasetV2* contains nearly 400,000 VM requests and covers approximately one month. To keep the workload scale comparable to *Huawei-East-1*, we use a chronological, contiguous subset: the first 200,000 VM requests ordered by time, which correspond to roughly a 9-day period. We retain only CPU and memory attributes

Figure EC.2 Characteristics of VM requests in Azure Dataset.

(a) The Number of VM Arrivals. (b) Average Duration Time. (c) Diverse Demand Size Distributions.



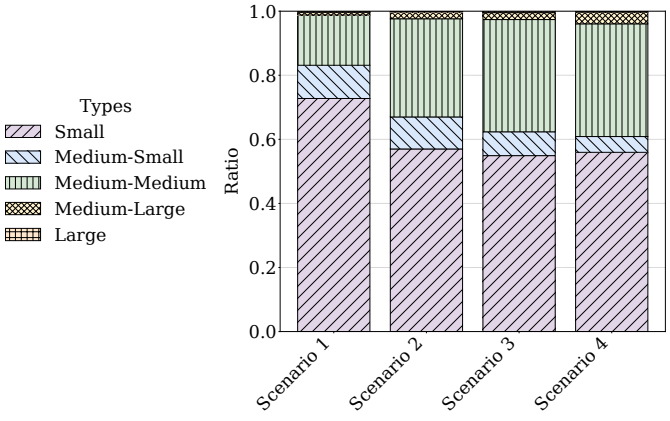
for Azure to enable a direct comparison of VM characteristics and performance metrics across the two datasets. Notably, we extract some VM sequences from the Azure dataset and find that these sequences lack sufficiently different scene features and are primarily composed of small and medium-sized requests, as shown in Figure EC.2. As this dataset contains only several days of VM request records and demonstrates no statistically significant nonstationary characteristics in request distribution, we partition it into four scenarios, as shown in Figure EC.3(a).

We train the MiCo algorithm on the Azure training set and evaluate it on the test set. Experimental results (detailed in Table EC.1) reveal two principal findings. First, MiCo consistently achieves superior scheduling performance across all scenarios. Particularly in Scenario 2, which features highly diverse VM requests, the algorithm exhibits remarkable effectiveness with a 17.3% performance gap compared to the heuristic Hindsight solution (70.3% vs. 53.0%). This represents an average improvement of about 6% over existing online scheduling algorithms, conclusively demonstrating the robustness advantages of MiCo in dynamic environments. Second, Scenarios 3 and 4 reveal systemic vulnerabilities when handling medium/large VM requests: Inappropriate resource reservation strategies increase placement failure probabilities, causing premature scheduling termination due to insufficient capacity for subsequent large-task allocations. But this may also be due to the fact that our experimental setup is limited to 50 PMs, which poses inherent challenges in reserving sufficient resources for sequential large-task allocations.

We further include a transfer setting, MiCo trained on Huawei and evaluated on Azure ($\text{MiCo}_{\text{transfer}}$), using the same experimental setup as the original Azure experiments, this appears as row 5 in Table EC.1 and performs poorly. The degradation stems from a data mismatch: as shown in Figure EC.3(a), Azure lacks the scenario structure present in Huawei, so a hierarchy mined on Huawei does not transfer well. In addition, we add Context-Independent FunSearch as a baseline on Azure, because Azure is relatively stationary, MiCo trained on Azure yields only marginal improvements over direct FunSearch, consistent with the limited

Figure EC.3 Scenario Features and Test-set Performance for Each Algorithm on the Azure Dataset.

(a) Scenario Features on the Azure Dataset.



(b) Scheduling Length Comparison on the Azure Test dataset.

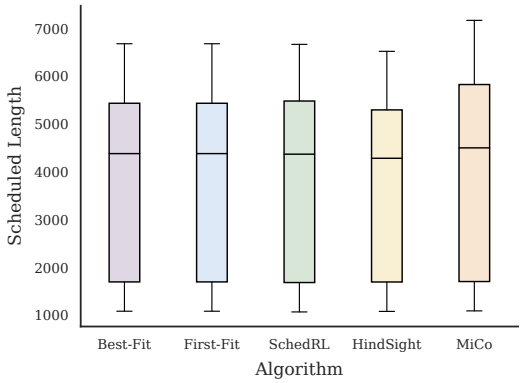


Table EC.1 Performance Comparison of Best-Fit, First-Fit, Hindsight, and MiCo Algorithms against Offline Solutions Obtained by Gurobi Across Scenarios in Azure Training Dataset.

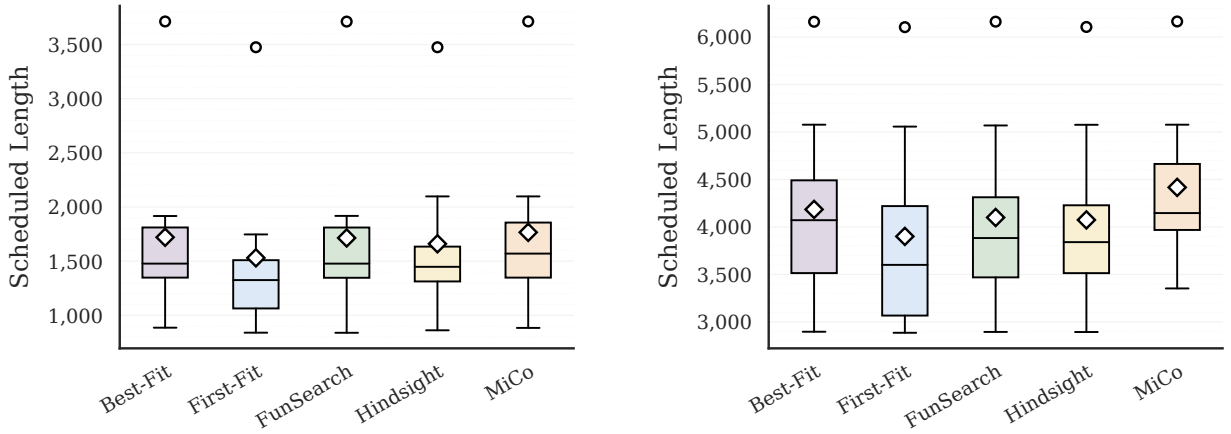
Algorithm \ Scenario	S^1	S^2	S^3	S^4	Mean
Best-Fit	99.2%	58.0%	57.5%	83.1%	65.5%
First-Fit	99.2%	58.2%	57.6%	83.1%	65.6%
Hindsight	99.2%	53.0%	58.2%	78.2%	62.8%
SchedRL	98.3(±0.3)%	58.1(±0.1)%	57.6(±0.2)%	83.5(±0.7)%	65.7(±0.3)%
$\text{MiCo}_{\text{transfer}}$	-	-	-	-	**63.5%**
FunSearch	-	-	-	-	**70.6%**
MiCo	**99.9%**	**70.3%**	**60.4%**	**86.5%**	**71.5%**

benefits of hierarchical composition in near-stationary settings. Additionally, we evaluate the algorithm on the test dataset. As shown in Figure EC.3(b), while the box plot shows similar median performance across all algorithms, the upper whisker of MiCo is longer and its box is positioned relatively higher. This further highlights the superiority of MiCo over the other algorithms.

We conduct an additional evaluation using a data-driven scenario construction and a chronological train-test split. The Huawei dataset is split chronologically, with the first 100,000 samples used for training and the subsequent 25,000 samples for testing. Scenarios are constructed on the training set by first learning a policy for each demand sequence independently, then evaluating these policies across sequences, and clustering them based on performance similarity. This procedure yields five representative scenarios. Figures EC.4(a) and EC.4(b) report test performance under 50-server and 100-server configurations, respectively. The policy trained under the 50-server setting is directly applied to the 100-server system without retraining. In both cases, our method consistently outperforms the baselines, demonstrating robustness to scenario construction and system scale.

Figure EC.4 Testset Performance for Each Algorithm on the Huawei Dataset.

(a) Performance in Huawei Test Dataset (50 servers). (b) Performance in Huawei Test Dataset (100 servers).



EC.3. Prompt Template

Scheduler Template

Given the existing priority_v0 function, please generate an optimized version named priority_v*. This new version should be more complex and efficient, incorporating multiple conditional logic and loops as necessary. The function should calculate priorities for items to be added to bins, considering the item size and bin capacities. Ensure the function is significantly different and more advanced than the prior versions. Only the Python code for the function is required, without any additional descriptions or annotations. Existing priority_v0 function for reference:

```
def priority_v0(bin, item):
    """ Calculate and return the priority score for adding a specific item to a bin based
        on available CPU and MEM resources.
    Args:
        bin (tuple): Tuple representing the bin's available resources, where bin[0] is CPU
            and bin[1] is memory.
        item (tuple): Tuple representing the item's resource requirements, where item[0] is
             CPU and item[1] is memory needed.
    Returns:
        int: The total score for placing the item in the current bin. A higher score
            indicates a better fit based on current available resources. """
    score = -(bin[0] - item[0])
    return score
def priority_v1(bin, item):
    """Improved version of `priority_v0`."""
```

Your task is to create the optimized priority_v* function based on the guidelines above. Remember, only the Python function code is needed.

Context-Aware Scheduler Template

You are a leading expert on this topic. Given the existing heuristic_selector_v0 function, please generate an optimized version named heuristic_selector_v*. This new version should be more complex and efficient, incorporating multiple conditional logic and loops as necessary. Ensure the function is significantly different and more advanced than the prior versions. Existing heuristic_selector_v0 function for reference:

```
def heuristic_selector_v0(condition):
    """ This function selects the appropriate heuristic scheduling function based on the
        input condition.param condition: list of dicts, each representing the distribution
        of request types over the past 200 requests, divided into 4 groups of 50 requests
        each.Each dictionary contains keys "small", "medium_small", "medium_medium", "
        medium_large", and "large"  with their respective proportions.Example:
    [{"small": 0.4, "medium_small": 0.3, "medium_medium": 0.1, "medium_large": 0.1, "large
        ": 0.1},
    {"small": 0.4, "medium_small": 0.3, "medium_medium": 0.1, "medium_large": 0.1, "large":
         0.1},
    {"small": 0.4, "medium_small": 0.3, "medium_medium": 0.1, "medium_large": 0.1, "large":
         0.1},
    {"small": 0.4, "medium_small": 0.3, "medium_medium": 0.1, "medium_large": 0.1, "large":
         0.1}]
    :return: int, index of the selected heuristic function (only 1,2,3,4)"""
    return 1
def heuristic_selector_v1(condition):
    """Improved version of `heuristic_selector_v0`."""
```

The “heuristic_selector" function should only return 1 or 2 or 3 or 4. In order to ensure the result, do not use any "random" in “heuristic_selector" function. Your task is to create the optimized heuristic_selector_v* function based on the guidelines above. Remember, only the Python function code is needed.

EC.4. Generated Heuristics

```python
def priority(bin, item):
    # Unpack CPU and memory resources
    cpu_resource, mem_resource = bin
    cpu_needed, mem_needed = item

    # Return -inf if item doesn't fit
    if cpu_needed > cpu_resource or mem_needed > mem_resource:
        return float('-inf')

    # ... Calculate various differences, ratios, and a weight_factor ...
    cpu_diff = cpu_resource - cpu_needed
    mem_diff = mem_resource - mem_needed
    ...
    weight_factor = 1 + ...

    # ... Loop to calculate the core min_weighted_diff ...
    min_weighted_diff = float('inf')
    for i in range(2):
        # ... Complex weighted_diff calculation ...
        weighted_diff = -((resource_diffs[0] + resource_diffs[1]) * ... * weight_factor)

        # ... Conditional adjustment ...
        if ...:
            weighted_diff *= ...

        min_weighted_diff = min(min_weighted_diff, weighted_diff)

    # ... Calculate remaining space and item/bin size ratio ...
    remaining_space = (cpu_diff * mem_diff) / (cpu_resource * mem_resource)
    min_weighted_diff *= (1 - remaining_space)

    ...
    size_ratio = (cpu_needed * mem_needed) / (cpu_resource * mem_resource)

    # ... Select weights based on size_ratio ...
    weighted_items = [0.7, 0.3] if size_ratio <= 0.5 else [0.6, 0.4]

    # ... Define helper function and calculate initial score ...
    def score_by_weight(weight):
        return min_weighted_diff * weight
    ...
    final_score = score_by_weight(best_weight)

    # ... Apply multiple adjustments, penalties, and bonuses to the final_score ...
    final_score += abs(size_ratio - 1) * 0.2
    if ...: # Penalty condition
        final_score *= ...
    if ...: # Bonus condition 1
        final_score *= ...
    if ...: # Bonus condition 2
        final_score *= ...
    final_score *= ... # Final factor adjustment

    # Return the computed final score
    return final_score
```

Listing 1: Heuristic Algorithm Generated by Option Miner

```python
def heuristic_selector(condition):
    import numpy as np

    # --- Step 1: Calculate statistics ---
    def calculate_stats(condition):
        stats = {
            "averages": {k: 0 for k in condition[0].keys()},
            "trends": {k: [] for k in condition[0].keys()},
            "accelerations": {k: [] for k in condition[0].keys()},
            "recent_changes": {k: 0 for k in condition[0].keys()},
            "recent_accelerations": {k: 0 for k in condition[0].keys()},
        }
        ...
        # compute averages, trends, accelerations
        ...
        return stats

    # --- Step 2: Weighted score calculation ---
    def calculate_weighted_scores(stats):
        weights = [0.25, 0.3, 0.3, 0.15]
        scores = {}
        for k in stats["averages"].keys():
            scores[k] = (
                weights[0] * stats["averages"][k]
                + weights[1] * np.std(stats["trends"][k]) / (np.mean(stats["trends"][k]) + 1e-6)
                + weights[2] * condition[-1][k]
                + weights[3] * np.std(stats["accelerations"][k]) / (np.mean(stats["accelerations"][k]) + 1e-6)
            )
        return scores
```

```
    stats = calculate_stats(condition)
    scores = calculate_weighted_scores(stats)

    # --- Step 3: Heuristic selection based on dominant metric ---
    dominant_request_type = max(scores, key=scores.get)
    heuristic_map = {"small": 1, "medium_small": 2, "medium_medium": 3, "medium_large": 4, "large": 4}
    selected_heuristic = heuristic_map.get(dominant_request_type, 2)

    # --- Step 4: Refinement rules ---
    if selected_heuristic == 4 and scores["medium_medium"] > scores["medium_small"]:
        selected_heuristic = 3
    elif selected_heuristic == 1:
        ...
    if stats["trends"]["large"][-1] > (np.mean(stats["trends"]["large"]) + 1.4 * np.std(stats["trends"]["large"])):
        selected_heuristic = 4

    recent_changes = np.array(list(stats["recent_changes"].values()))
    if np.sum(recent_changes ** 2) > 1.2 * len(recent_changes):
        ...

    recent_accelerations = np.array(list(stats["recent_accelerations"].values()))
    if np.sum(recent_accelerations ** 2) > 1.5 * len(recent_accelerations):
        ...

    balanced = [abs(scores[k] - scores["large"]) < 0.1 for k in scores.keys() if k != "large"]
    if all(balanced):
        ...

    if selected_heuristic < 1 or selected_heuristic > 4:
        selected_heuristic = 2

    return selected_heuristic
```

Listing 2: Heuristic Algorithm Generated by Option Composer

References

AhmadiTeshnizi, A., W. Gao, and M. Udell (2024). OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. arXiv preprint arXiv:2402.10172.

Azar, Y., I. R. Cohen, S. Kamara, and B. Shepherd (2013). Tight bounds for online vector bin packing. In Proceedings of the 45th annual ACM Symposium on Theory of Computing, pp. 961–970.

Azar, Y. and D. Vainstein (2019). Tight bounds for clairvoyant dynamic bin packing. ACM Transactions on Parallel Computing 6(3), 1–21.

Bentley, J. L., D. S. Johnson, F. T. Leighton, C. C. McGeoch, and L. A. McGeoch (1984). Some unexpected expected behavior results for bin packing. In Proceedings of the 16th Annual ACM Symposium on Theory of Computing, pp. 279–288.

Berg, B. P. and B. T. Denton (2017). Fast approximation methods for online scheduling of outpatient procedure centers. INFORMS Journal on Computing 29(4), 631–644.

Cai, Q., W. Hang, A. Mirhoseini, G. Tucker, J. Wang, and W. Wei (2019). Reinforcement learning driven heuristic optimization. arXiv preprint arXiv:1906.06639. DRL4KDD'19.

Chen, S., K. Moinzadeh, J.-S. Song, and Y. Zhong (2023). Cloud computing value chains: Research from the operations management perspective. Manufacturing & Service Operations Management 25(4), 1338–1356.

Christensen, H. I., A. Khan, S. Pokutta, and P. Tetali (2017). Approximation and online algorithms for multidimensional bin packing: A survey. Computer Science Review 24, 63–79.

Coffman, E. G., M. R. Garey, and D. S. Johnson (1983). Dynamic bin packing. SIAM Journal on Computing 12(2), 227–258.

Coffman, E. G., M. R. Garey, and D. S. Johnson (1984). Approximation algorithms for bin-packing–an updated survey. In Algorithm Design for Computer System Design, pp. 49–106.

Coffman, E. G. and A. L. Stolyar (2001). Bandwidth packing. Algorithmica 29, 70–88.

Côté, J.-F., M. Haouari, and M. Iori (2021). Combinatorial benders decomposition for the two-dimensional bin packing problem. INFORMS Journal on Computing 33(3), 963–978.

Garey, M. R., R. L. Graham, D. S. Johnson, and A. C.-C. Yao (1976). Resource constrained scheduling as generalized bin packing. Journal of Combinatorial Theory, Series A 21(3), 257–298.

Gurobi Optimization, LLC (2024). Gurobi Optimizer Reference Manual.

Hadary, O., L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, et al. (2020). Protean:VM allocation service at scale. In The 14th USENIX Symposium on Operating Systems Design and Implementation, pp. 845–861.

Huang, C., Z. Tang, S. Hu, R. Jiang, X. Zheng, D. Ge, B. Wang, and Z. Wang (2025). ORLM: A customizable framework in training large models for automated optimization modeling. Operations Research 73(6), 2986–3009.

Jiang, Y., Z. Cao, and J. Zhang (2021). Learning to solve 3D bin packing problem via deep reinforcement learning and constraint programming. IEEE Transactions on Cybernetics 53(5), 2864–2875.

Maguluri, S. T., R. Srikant, and L. Ying (2012). Stochastic models of load balancing and scheduling in cloud computing clusters. In The 31st Annual IEEE International Conference on Computer Communications, pp. 702–710.

Martello, S., D. Pisinger, and D. Vigo (2000). The three-dimensional bin packing problem. Operations Research 48(2), 256–267.

Pisinger, D. and M. Sigurd (2007). Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. INFORMS Journal on Computing 19(1), 36–51.

Puchinger, J. and G. R. Raidl (2007). Models and algorithms for three-stage two-dimensional bin packing. European Journal of Operational Research 183(3), 1304–1327.

Ramamonjison, R., T. Yu, R. Li, H. Li, G. Carenini, B. Ghaddar, S. He, M. Mostajabdaveh, A. Banitalebi-Dehkordi, Z. Zhou, et al. (2023). NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In NeurIPS 2022 Competition Track, pp. 189–203.

Romera-Paredes, B., M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. (2024). Mathematical discoveries from program search with large language models. Nature 625(7995), 468–475.

Stolyar, A. L. (2013). An infinite server system with general packing constraints. Operations Research 61(5), 1200–1217.

Stolyar, A. L. and Y. Zhong (2021). A service system with packing constraints: Greedy randomized algorithm achieving sublinear in scale optimality gap. Stochastic Systems 11(2), 83–111.

Ye, H., J. Wang, Z. Cao, F. Berto, C. Hua, H. Kim, J. Park, and G. Song (2024). ReEvo: Large language models as hyper-heuristics with reflective evolution. arXiv preprint arXiv:2402.01145. Accepted at NeurIPS 2024.

Zhang, Y., R. Bai, R. Qu, C. Tu, and J. Jin (2022). A deep reinforcement learning based hyper-heuristic for combinatorial optimisation with uncertainties. European Journal of Operational Research 300(2), 418–427.

Zhang, Z., B. T. Denton, and X. Xie (2020). Branch and price for chance-constrained bin packing. INFORMS Journal on Computing 32(3), 547–564.

Zhao, H., Q. She, C. Zhu, Y. Yang, and K. Xu (2021). Online 3D bin packing with constrained deep reinforcement learning. In Proceedings of the AAAI Conference on Artificial Intelligence, Volume 35, pp. 741–749.