Approximating Solutions to the Knapsack Problem Using Machine Learning and Genetic Algorithms

Aaditya S Rao[1]
[1]Department of Information Science and Engineering
R. V. College Of Engineering
Bengaluru, India
aadityasrao.is23@rvce.edu.in

Aditi Shastri[2]
[2]Department of Information Science and Engineering
R. V. College Of Engineering
Bengaluru, India
aditishastri.is23@rvce.edu.in

Asish Kumar Yeleti[3]
[3]Department of Information Science and Engineering
R. V. College Of Engineering
Bengaluru, India
asishkumary.is23@rvce.edu.in

Prof. Swetha S[4]
[4]Department of Information Science and Engineering
R. V. College Of Engineering
Bengaluru, India
swetas.ise@rvce.edu.in

***Abstract*—The knapsack problem is a classic combinatorial optimization challenge with applications across numerous domains. While traditional algorithms provide exact solutions, they often become computationally intensive for large problem instances. This paper presents multiple machine learning approaches to efficiently approximate optimal solutions to the knapsack problem. We explore the effectiveness of supervised learning models (Random Forest, Gradient Boosting, and Neural Networks) as well as evolutionary computation through Genetic Algorithms. Our hybrid approach combines ML predictions with solution repair mechanisms and local search optimization to maintain solution feasibility and quality. Experimental results demonstrate that our models achieve near-optimal solutions with significantly reduced computational overhead compared to exact algorithms. The integrated web platform provides an interactive environment for comparing different solution approaches, offering insights into the trade-offs between solution quality and computational efficiency.**

***Index Terms*—knapsack problem, machine learning, genetic algorithms, combinatorial optimization, approximation algorithms, neural networks, random forest, gradient boosting**

I. INTRODUCTION

The knapsack problem is a fundamental combinatorial optimization problem that has attracted significant attention in computer science, operations research, and applied mathematics for decades. Formally introduced in the early 20th century, this problem elegantly captures the essence of resource allocation under constraints. Given a set of items, each with a weight and value, the problem asks to determine which items to include in a collection such that the total weight is less than or equal to a given limit (capacity) and the total value is maximized.

Despite its simple formulation, the knapsack problem appears in numerous real-world applications spanning diverse domains. In finance, portfolio optimization involves selecting investments within budget constraints to maximize returns. In logistics, cargo loading requires selecting shipments to maximize value while respecting weight or volume limits. In computational biology, the problem manifests in selecting genetic markers or protein sequences under experimental constraints. In cloud computing, resource allocation decisions involve maximizing service quality while respecting hardware limitations. Even in project selection scenarios, managers must choose projects to maximize benefits while staying within budget constraints.

The widespread applicability of the knapsack problem has led to extensive research on solution methodologies. The classic formulation of the 0-1 knapsack problem is NP-hard, indicating that no known polynomial-time algorithm can guarantee an optimal solution for all instances. Traditional exact algorithms include dynamic programming approaches that run in pseudo-polynomial time O(nW), where n is the number of items and W is the knapsack capacity. Branch and bound methods offer improvements in practice but still face exponential worst-case complexity. For large-scale instances with hundreds or thousands of items or when capacity values are large, these approaches become computationally infeasible, often requiring minutes or even hours to compute optimal solutions.

Approximation algorithms, such as the fully polynomial-time approximation scheme (FPTAS), provide theoretical guarantees on solution quality while improving computational efficiency. However, these algorithms typically involve complex implementations and parameter tuning that can limit their practical adoption. Greedy heuristics, such as sorting items by value-to-weight ratio, offer fast solutions but can produce results far from optimal in many cases, particularly when items have complex relationships or similar ratios.

Recent advances in machine learning provide opportunities to approach combinatorial optimization problems from a new perspective: learning from existing solutions instead of solving from first principles. The field of machine learning for combinatorial optimization, as surveyed by Bengio et al. [2],

has shown promising results across various NP-hard problems. This approach offers the potential to capture complex patterns in optimal solutions and generalize to new problem instances without the computational overhead of traditional algorithms.

Concurrently, evolutionary computation approaches, particularly genetic algorithms, have demonstrated effectiveness in navigating large solution spaces for combinatorial problems. By mimicking natural selection processes, these methods can explore diverse solution candidates and progressively improve quality through operations like crossover and mutation.

Our research explores three complementary approaches to the knapsack problem, each with distinct advantages and limitations:

First, we develop supervised machine learning models that learn to predict item selections based on problem instance features. By training on datasets of optimal solutions, these models can rapidly approximate solutions to new problems without having to execute complex algorithms. We compare three different architectures: Random Forests for interpretability, Gradient Boosting for accuracy, and Neural Networks for handling complex feature interactions.

Second, we implement a genetic algorithm framework that evolves populations of candidate solutions through selection, crossover, and mutation operations. This approach is particularly effective for exploring large solution spaces and avoiding local optima that might trap deterministic algorithms.

Third, we propose a novel hybrid approach that combines machine learning predictions with solution repair mechanisms and local search optimization. This approach leverages the strengths of both paradigms: the speed of ML inference and the solution quality improvements of local search techniques.

Our implementation, KnapsackML, provides an integrated platform for comparing these approaches across different problem sizes and distributions. The system includes a modern web interface for interactive exploration, allowing users to visualize solution quality, computational efficiency, and algorithm behavior. We conduct extensive experiments to evaluate performance across diverse problem instances, analyzing the trade-offs between solution quality and computational resources.

By addressing the knapsack problem through multiple paradigms, our research contributes to the growing field of learning-based approaches to combinatorial optimization. The insights gained here may extend to other NP-hard problems where similar hybrid approaches could balance solution quality and computational efficiency.

The remainder of this paper is organized as follows: Section II reviews related work in combining machine learning with combinatorial optimization. Section III details our proposed methodology, including data generation, model architectures, and training processes. Section IV presents experimental results and analysis, while Section V concludes with a discussion of limitations and future work.

II. RELATED WORK

A. Traditional Approaches to the Knapsack Problem

The knapsack problem has been extensively studied in operations research and computer science for more than a century. The problem's theoretical properties, complexity, and solution approaches have been thoroughly analyzed in comprehensive texts such as those by Kellerer et al. [1] and Martello and Toth [4].

Dynamic programming, first formalized by Bellman [3], remains the cornerstone exact algorithm for the knapsack problem. This approach builds a two-dimensional table of size O(nW), where each cell represents the maximum value achievable using a subset of the first i items with capacity j. While this method guarantees optimal solutions, its pseudo-polynomial time complexity becomes prohibitive for large instances. Pisinger [18] introduced improvements to dynamic programming through bound-based methods that reduce the effective state space, achieving significant speedups while maintaining optimality.

Branch and bound algorithms, as developed by Horowitz and Sahni [12], address the exponential complexity by intelligently pruning the search space. These methods establish upper and lower bounds for partial solutions, discarding branches that cannot lead to improvements over the best-known solution. Martello and Toth [4] further refined these techniques with improved bounding procedures and variable ordering heuristics. Pisinger [18] developed linear time algorithms for specific weight-bounded knapsack variants. Modern implementations like the one by Pisinger [8] can efficiently solve problems with thousands of items in specific distributions.

Approximation algorithms provide theoretical guarantees on solution quality while improving computational efficiency. The fully polynomial-time approximation scheme (FPTAS) developed by Ibarra and Kim [5] achieves (1-)-optimal solutions in time polynomial in both n and 1/, offering a tunable trade-off between solution quality and computational effort. Practical implementations of FPTAS, however, often involve complex parameter tuning that limits their adoption in time-sensitive applications.

Greedy heuristics, particularly the value-to-weight ratio sorting approach, offer simple and fast approximations but can perform poorly on carefully constructed instances. Nevertheless, they remain valuable as initialization methods for more sophisticated algorithms or when computational resources are severely constrained.

B. Machine Learning for Combinatorial Optimization

The integration of machine learning with combinatorial optimization represents an emerging paradigm that has gained significant momentum in recent years. Bengio et al. [2] provide a comprehensive survey of this field, categorizing approaches into learning to optimize directly and learning to configure optimization algorithms.

For direct optimization approaches, early work by Vinyals et al. [17] introduced Pointer Networks, which use attention mechanisms to learn constructive heuristics for combinatorial problems. Subsequent research by Bello et al. [9] combined these networks with reinforcement learning to improve solution quality. Kool et al. [7] further advanced this approach with their attention model that outperformed classical heuristics on routing problems.

Graph neural networks (GNNs) have emerged as particularly promising for combinatorial optimization due to their ability to capture structural relationships. Li et al. [19] demonstrated that GNNs combined with guided tree search could effectively tackle maximum cut and minimum vertex cover problems. Cappart et al. [14] provided a comprehensive overview of how these approaches can be applied across different optimization domains, showing competitive performance with specialized heuristics at reduced computational cost.

For the knapsack problem specifically, recent research has employed neural networks to learn effective construction heuristics. Delarue et al. [20] explored reinforcement learning for combinatorial optimization, applying their approach to knapsack variants. Mazyavkina et al. [15] provided a comprehensive survey of reinforcement learning applications in this domain, showing that agents can learn effective policies through interaction with problem environments. These works demonstrate that networks trained on optimal solutions could generalize to unseen problem instances.

A significant advantage of learning-based approaches is their ability to amortize computation time across problem instances. While traditional algorithms must solve each instance from scratch, learned models can leverage patterns from previously solved problems to rapidly approximate solutions to new instances. However, generalization across diverse problem distributions remains challenging, as noted by Khalil et al. [10] in their work on learning combinatorial optimization over graphs.

C. Evolutionary Approaches for Combinatorial Optimization

Evolutionary computation offers a powerful framework for addressing complex combinatorial optimization problems through biologically-inspired search mechanisms. Genetic algorithms (GAs), first introduced by Holland [16] and popularized for optimization by Goldberg [11], have been successfully applied to numerous NP-hard problems.

For the knapsack problem, Chu and Beasley [6] developed a genetic algorithm for the multidimensional variant that remains a benchmark in the field. Their approach incorporated problem-specific repair operators to maintain feasibility while preserving solution diversity. Raidl [13] further improved this methodology by introducing greedy preprocessing and local search to enhance solution quality, demonstrating the effectiveness of hybridizing evolutionary computation with problem-specific heuristics.

One of the key advantages of genetic algorithms is their inherent parallelism, allowing them to simultaneously explore multiple regions of the solution space. This property makes them particularly suited for complex landscapes with multiple local optima that might trap deterministic methods. Pisinger [8] explored challenging problem instances where traditional algorithms struggle but evolutionary approaches can effectively navigate the irregular search space of constrained combinatorial problems.

The performance of genetic algorithms depends critically on their configuration. Extensive research has focused on optimal population sizing, selection mechanisms, and genetic operator design. For the knapsack problem, tournament selection has proven effective due to its adjustable selection pressure, while single-point crossover maintains building blocks of high-quality solutions.

Research in evolutionary computation has introduced adaptive parameter control mechanisms that dynamically adjust mutation rates and selection pressure based on population diversity measures. This approach addresses the exploration-exploitation trade-off that is fundamental to evolutionary search, allowing algorithms to automatically balance between finding new promising regions and refining solutions in currently known regions.

The integration of problem-specific knowledge into genetic algorithms has been another productive research direction. Chu and Beasley [6] developed specialized operators specifically designed for knapsack problems that preserve high value-to-weight ratio items, significantly improving convergence rates. Similarly, Horowitz and Sahni [12] provided insights that can be applied to initialize populations with heuristic solutions rather than random selections, substantially accelerating evolutionary optimization.

Our work builds upon these foundations and introduces a novel hybrid approach that combines supervised learning from optimal solutions with evolutionary optimization and local search techniques. Unlike previous methods that typically use either learning-based or evolutionary approaches in isolation, our framework integrates these paradigms to exploit their complementary strengths, addressing both solution quality and computational efficiency considerations.

III. METHODOLOGY

A. Problem Formulation

The 0-1 knapsack problem is defined as follows: Given n items, each with a weight w_i and value v_i, and a knapsack capacity C, select a subset of items to maximize the total value while keeping the total weight within capacity. This can be formulated as:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq C \tag{2}$$

$$x_i \in \{0,1\}, \forall i \in \{1,2,\ldots,n\} \tag{3}$$

where x_i is a binary variable indicating whether item i is selected (1) or not (0).

This formulation captures the essence of resource allocation under constraints. The binary decision variables reflect the discrete nature of selection decisions—each item is either fully included or excluded without the possibility of fractional selections. The objective function represents the cumulative value of selected items, while the constraint ensures that the total weight remains within the specified capacity. This problem formulation applies directly to scenarios where resources cannot be divided, such as project selection, cargo loading, and digital storage allocation.

In our research, we explore three distinct but complementary approaches to solve this problem. The first approach leverages supervised machine learning techniques to learn patterns from optimal solutions. This methodology transforms the combinatorial optimization problem into a pattern recognition task, where models are trained to predict item selection decisions based on problem instance characteristics. By learning from a diverse set of pre-solved instances, these models can generalize to new problems without executing computationally intensive algorithms.

The second approach utilizes evolutionary computation through genetic algorithms. This population-based metaheuristic mimics natural selection by evolving sets of candidate solutions across generations. Through mechanisms of selection, crossover, and mutation, the algorithm progressively improves solution quality while maintaining diversity in the search process. This approach is particularly effective for exploring large solution spaces and escaping local optima.

Our third approach is a novel hybrid methodology that combines machine learning predictions with solution repair mechanisms and local search optimization. This integration addresses the complementary weaknesses of each method: machine learning models may produce infeasible solutions that violate constraints, while local search methods may struggle with initialization and global exploration. By combining these techniques, we achieve both computational efficiency and solution quality.

B. System Architecture and Methodology Overview

Fig. 1 presents the comprehensive flowchart of our methodology, illustrating the integration of machine learning, genetic algorithms, and hybrid optimization approaches for solving the knapsack problem.

The system architecture consists of the following key components and data flows:

1) **Problem Instance Generation** - The system begins by creating or receiving knapsack problem instances, characterized by item weights, values, and capacity constraints. For training purposes, we generate diverse problem distributions to ensure robust model learning.
2) **Feature Extraction** - Raw problem data undergoes comprehensive feature engineering, where statistical properties of weights, values, and value-to-weight ratios are computed to create a rich feature vector representing each instance.
3) **Solution Pathway Selection** - Based on problem characteristics and user preferences, the system determines whether to employ machine learning prediction, genetic algorithm, or the hybrid approach.
4) **Machine Learning Component** - This pathway utilizes trained models (Random Forest, Gradient Boosting, or Neural Network) to predict item selections based on extracted features, providing rapid initial solutions.
5) **Genetic Algorithm Component** - This evolutionary pathway evolves populations of candidate solutions through selection, crossover, and mutation operations, progressively improving solution quality.
6) **Hybrid Optimization Component** - The hybrid approach combines ML predictions with solution repair mechanisms and local search optimization to ensure feasibility and enhance solution quality.
7) **Solution Evaluation** - All generated solutions undergo evaluation for quality (total value), feasibility (capacity constraint satisfaction), and computational efficiency (time and memory usage).
8) **Results Visualization** - The final component presents solution metrics and visualizations to facilitate analysis and comparison between different approaches.

This integrated architecture enables systematic comparison of solution approaches across different problem instances, providing insights into the strengths and limitations of each methodology. The modular design facilitates extension with new algorithms or problem variants, supporting ongoing research in combinatorial optimization techniques.

C. Machine Learning Models

1) Feature Engineering: Feature engineering is a critical component of our machine learning approach. Effective representation of knapsack problem instances directly influences model performance and generalization capability. Our feature engineering process transforms the raw problem data into a rich set of descriptive metrics that capture the underlying structure and characteristics of each instance.

For each problem instance, we extract multiple feature categories that together provide a comprehensive representation. The first category comprises basic problem statistics including the number of items, knapsack capacity, total weight of all items, and total value of all items. These features establish the fundamental scale and constraints of the problem instance.

Statistical properties of weights form the second category, encompassing mean, standard deviation, median, first quartile (25th percentile), and third quartile (75th percentile). These metrics characterize the distribution of item weights, indicating whether weights are uniformly distributed or skewed, and whether the distribution contains outliers that might significantly impact solution strategies.

Similarly, the third category includes statistical properties of values: mean, standard deviation, median, and quartiles. These metrics provide insights into the value distribution, helping models identify instances where a few high-value

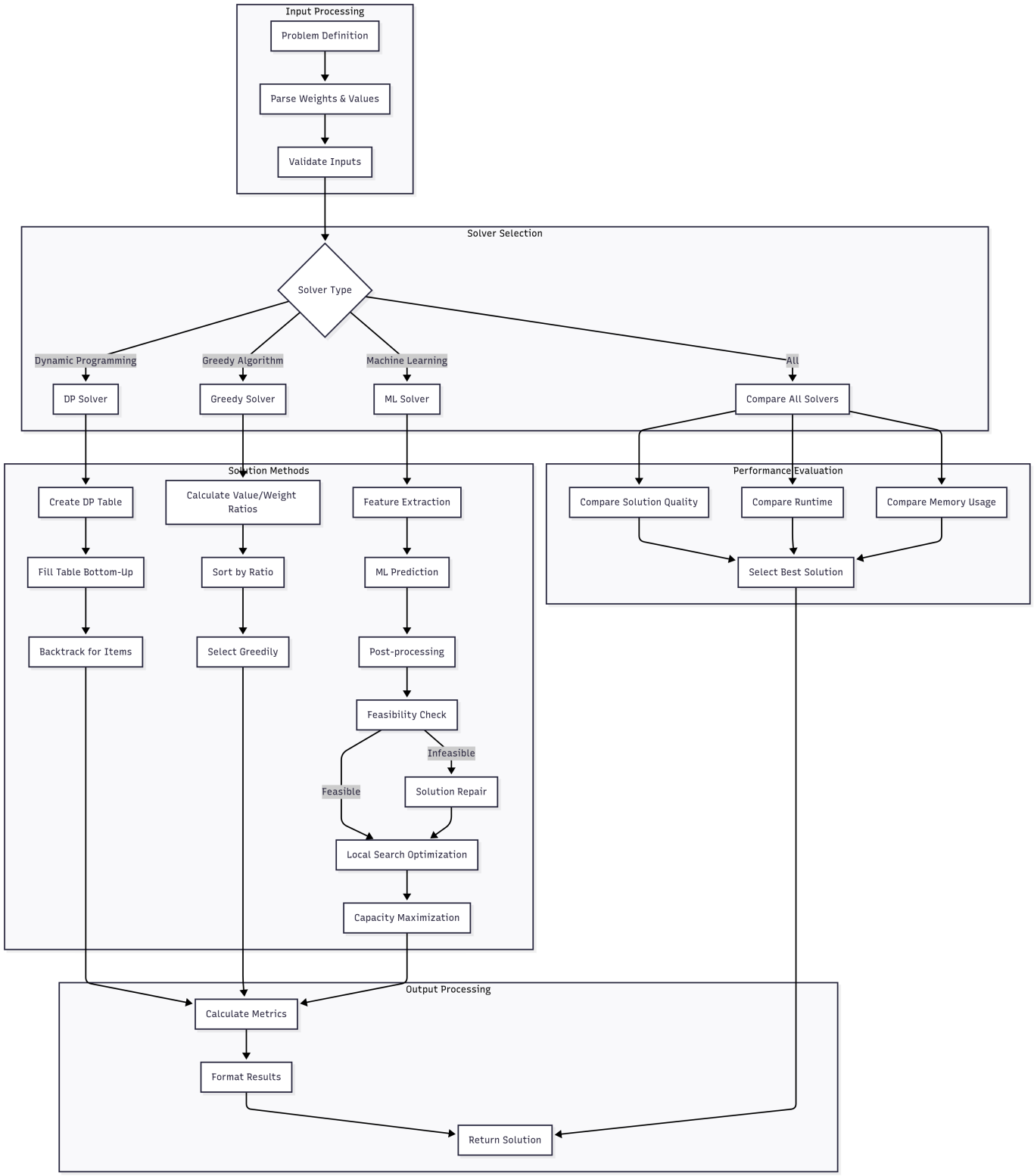

Fig. 1. Methodology flowchart showing the integrated approach to knapsack problem solving. The system combines ML prediction, solution repair, local search optimization and genetic algorithm components.

items dominate the solution space versus scenarios where values are more evenly distributed across items.

The fourth category focuses on value-to-weight ratios, which are particularly informative for knapsack problems since they approximate the "efficiency" of each item. We include the mean, standard deviation, median, minimum, and maximum ratios. Additionally, we extract the top-k and bottom-k ratios (where k=5 in our implementation) to highlight the most and least efficient items in the problem instance.

Advanced features constitute our fifth category, including knapsack density (the ratio of capacity to total weight), which indicates how selective the solution must be, and the correlation coefficient between weights and values, which reveals whether heavier items tend to be more valuable or if the relationship is random or inverse.

The final category encompasses higher-order statistics such as skewness and kurtosis of weights, values, and value-to-weight ratios. Skewness measures the asymmetry of the distributions, while kurtosis captures the "tailedness" of the distributions. These metrics help identify instances with unusual distribution shapes that might require specific solution strategies.

All features are standardized using a standard scaler that transforms each feature to have zero mean and unit variance. This normalization is essential for ensuring consistent performance across different problem distributions and prevents features with larger numerical ranges from dominating the learning process. The resulting feature vectors provide a rich, normalized representation of each knapsack problem instance that enables effective learning and generalization.

2) Model Architectures: We evaluate three distinct supervised learning architectures, each offering different trade-offs between interpretability, accuracy, and computational requirements.

Random Forest models serve as our first architecture, implemented as an ensemble of 500 decision trees with a maximum depth of 40. This configuration balances model complexity with generalization capability. We employ the CART (Classification and Regression Trees) algorithm with balanced class weights to address potential imbalances in the dataset, particularly when certain items are rarely selected in optimal solutions. Random Forests offer several advantages for our application: they handle non-linear relationships effectively, provide interpretable feature importance metrics, and are robust against overfitting due to their ensemble nature. Each tree in the forest is trained on a bootstrap sample of the training data, and final predictions are determined by majority voting across all trees, resulting in stable and reliable predictions.

Gradient Boosting models constitute our second architecture, comprising an ensemble of 300 gradient-boosting decision trees. We configure these models with a learning rate of 0.1 and maximum tree depth of 8, parameters determined through cross-validation to optimize performance while preventing overfitting. Since knapsack problems require binary decisions for each item, we wrap the gradient boosting classifier with a multi-output classifier to handle simultaneous prediction of multiple binary decisions. Gradient Boosting offers distinct advantages over Random Forests, particularly in handling complex patterns and achieving higher prediction accuracy through its sequential training process where each tree corrects errors made by previous trees.

Our third architecture employs Multi-Layer Perceptron (MLP) neural networks with three hidden layers containing 512, 256, and 128 neurons respectively. This architecture progressively reduces dimensionality while extracting increasingly abstract features from the input data. We use Rectified Linear Units (ReLU) as the activation function for all hidden layers, which helps mitigate the vanishing gradient problem while maintaining computational efficiency. The network is trained using the Adam optimizer with an adaptive learning rate scheduling mechanism that reduces the learning rate when performance plateaus. Early stopping with a patience of 15 epochs prevents overfitting by monitoring validation performance. The neural network approach offers superior capability in modeling complex non-linear relationships and interactions between features, though at the cost of reduced interpretability compared to tree-based models.

Each model architecture is trained using binary cross-entropy loss, which is appropriate for the binary decision nature of the knapsack problem. During inference, the models produce a probability for each item indicating its likelihood of inclusion in the optimal solution. These probabilities are thresholded at 0.5 to obtain binary decisions, which may require subsequent feasibility repair if they violate the capacity constraint.

D. Genetic Algorithm

Our genetic algorithm implementation follows a standard evolutionary approach that mimics natural selection processes to progressively improve solution quality. The core algorithm is outlined in Algorithm 1, which provides a high-level view of the optimization process.

Algorithm 1 Knapsack Genetic Algorithm

```
0: procedure SolveKnapsackGA(weights, values, capacity, population_size, generations, mutation_rate, elite_size)
0:     Initialize population with random binary strings
0:     for g = 1 to generations do
0:         Calculate fitness for each solution
0:         Select parents using tournament selection
0:         Create offspring through single-point crossover
0:         Apply mutation with probability mutation_rate
0:         Preserve elite_size best solutions
0:         Form new population from offspring and elites
0:     end for
0:     return best solution found
0: end procedure=0
```

Key components of the genetic algorithm include:

The population initialization process creates an initial set of candidate solutions represented as binary strings of length n, where n is the number of items. Each bit position corresponds to a specific item, with a value of 1 indicating selection and 0 indicating exclusion. We generate these strings randomly to ensure diversity in the starting population, which is crucial for effective exploration of the solution space. This diversity prevents premature convergence to suboptimal solutions by providing a wide range of genetic material for subsequent evolutionary operations. The population size is configurable, with larger populations providing more thorough exploration at the cost of increased computational overhead.

Our fitness function evaluates solution quality by calculating the total value of selected items while enforcing the capacity constraint. For feasible solutions (those not exceeding capacity), fitness equals the sum of values of selected items. For infeasible solutions, we apply a penalty proportional to the degree of constraint violation, allowing borderline infeasible solutions to remain in the population during early generations while gradually steering evolution toward feasible regions. This soft constraint handling approach enables more effective exploration than immediate rejection of all infeasible solutions, particularly in tightly constrained problem instances.

Selection is implemented using tournament selection with a tournament size of 5, balancing selection pressure and population diversity. In each tournament, five individuals are randomly sampled from the population, and the one with the highest fitness is selected as a parent. This process is repeated to select pairs of parents for reproduction. Tournament selection provides several advantages over alternative methods: it doesn't require global fitness ranking, can be easily parallelized, and allows tuning of selection pressure through the tournament size parameter. When compared to fitness-proportional selection, it also mitigates scaling issues when fitness values have widely varying magnitudes.

Single-point crossover creates offspring by combining genetic material from two parent solutions. A random crossover point is selected, and two children are produced by exchanging the portions of the parents' binary strings after that point. This operation preserves valuable building blocks (consecutive sequences of bits) that may represent effective partial solutions. The ability to exchange large sections of genetic material allows the algorithm to combine complementary strengths from different solutions, accelerating progress toward high-quality solutions compared to methods that rely solely on incremental improvements.

Mutation introduces small random variations by flipping bits in offspring solutions with probability equal to the mutation rate parameter. This operation serves multiple purposes: it maintains genetic diversity throughout the evolutionary process, helps escape local optima, and enables exploration of solution regions not accessible through crossover alone. The mutation rate requires careful tuning, as excessive mutation can disrupt beneficial gene combinations, while insufficient mutation can lead to premature convergence. Our implementation uses an adaptive mutation rate that decreases as the population converges, balancing exploration in early generations with exploitation in later ones.

Elitism preserves a predefined number (elite_size) of the best solutions from each generation, ensuring that high-quality genetic material is never lost due to stochastic operations. These elite solutions pass unchanged to the next generation and can continue to participate in reproduction, providing a stable foundation for further improvement. By maintaining the best-found solutions, elitism accelerates convergence and improves the final solution quality. In our implementation, the elite size is typically set to 10-20% of the population size, striking a balance between stability and population renewal.

The final population for each new generation is formed by combining the offspring produced through crossover and mutation with the preserved elite solutions. This process continues for a specified number of generations or until a termination criterion is met, such as no improvement for a certain number of consecutive generations. Upon completion, the algorithm returns the best solution found during the entire evolutionary process.

One key advantage of our genetic algorithm implementation is its ability to handle large problem instances without the memory constraints that affect dynamic programming approaches. Additionally, the population-based nature of the search allows parallel exploration of multiple promising regions in the solution space, increasing the likelihood of finding high-quality solutions even in complex problem landscapes.

E. Hybrid Approach and Solution Repair

Our hybrid approach combines ML predictions with solution repair and local optimization techniques:

Algorithm 2 Hybrid ML-Optimization Approach

```
0: procedure HybridKnapsackSolver(weights, values, capacity, model)
0:     Extract features from problem instance
0:     Get initial selection prediction from ML model
0:     if solution exceeds capacity then
0:         Repair solution by removing items (lowest value/weight ratio first)
0:     end if
0:     Apply local search optimization
0:     Maximize capacity utilization with remaining capacity
0:     return optimized solution
0: end procedure=0
```

The solution repair mechanisms include:

Feasibility repair is a critical component that addresses the inherent limitation of machine learning models in respecting hard constraints. When an ML model predicts a solution that exceeds the knapsack capacity, this mechanism systematically removes items until the solution becomes feasible. Items are removed in ascending order of their value-to-weight ratio, prioritizing the removal of less efficient items that contribute less value per unit of weight consumed. This greedy approach

minimizes the potential value loss during the repair process. Our implementation maintains a priority queue of selected items sorted by their value-to-weight ratios, enabling efficient identification of candidates for removal. The repair process terminates as soon as the capacity constraint is satisfied, ensuring minimal modification to the original prediction. This repair mechanism is particularly important for densely packed instances where the optimal solution approaches the capacity limit, as even small prediction errors can lead to constraint violations.

Local search optimization enhances solution quality by systematically exploring the neighborhood of the repaired solution. Starting from a feasible solution, the algorithm evaluates potential single-bit modifications—either adding an excluded item or removing an included item—to identify improvements in total value while maintaining feasibility. We implement a first-improvement strategy that applies beneficial modifications as soon as they are found, rather than evaluating the entire neighborhood before making changes. This approach accelerates the search process while still discovering significant improvements. The neighborhood exploration follows a value-to-weight ratio ordering: when considering item additions, we prioritize high-ratio items; when considering removals, we prioritize low-ratio items. This guided search substantially reduces the effective neighborhood size while focusing on the most promising modifications. The local search continues until no further single-bit improvement is possible, indicating that a local optimum has been reached. Multiple random restarts with perturbations are employed to escape local optima, allowing the algorithm to explore different regions of the solution space.

Capacity maximization serves as the final refinement step, addressing potential inefficiencies where valuable capacity remains unused after the previous optimization steps. This process identifies the remaining (unused) capacity in the knapsack and attempts to fill it with additional items that were previously excluded. Rather than simply considering items in descending value-to-weight ratio order, our implementation employs a dynamic programming approach on the subset of excluded items with the remaining capacity. This approach finds the optimal combination of excluded items that can fit in the residual capacity, maximizing the additional value contribution. The dynamic programming table is constructed incrementally by considering each excluded item and updating the maximum achievable value for each possible remaining capacity value. Once the table is complete, we backtrack to determine which items should be added to the solution. This capacity maximization step ensures that no capacity is wasted when it could be used to increase the solution value, addressing a common limitation of greedy approaches that cannot reconsider earlier decisions once made.

F. Data Generation and Training

To train our ML models, we generate a comprehensive dataset of knapsack problem instances with varying characteristics. The dataset generation process is designed to cover diverse problem distributions, ensuring that our models generalize well to a wide range of real-world scenarios.

We systematically vary problem sizes from 10 to 50 items to train models capable of handling different scales. This range was chosen based on preliminary experiments that showed diminishing returns in prediction accuracy beyond 50 items due to the increasing complexity of the solution space. For each problem size, we generate thousands of instances to ensure sufficient representation across the distribution space. The number of instances increases with problem size, with approximately 1,000 instances for 10-item problems and 10,000 instances for 50-item problems, compensating for the exponential growth in potential solution combinations.

Weights and values are sampled from multiple probability distributions to ensure robustness to different data characteristics. We include uniform distributions (U[1, 100]), normal distributions (N(50, 20)), exponential distributions (Exp(0.02)), and log-normal distributions (LogN(3, 1)). Additionally, we create instances with correlated weights and values using controlled correlation coefficients ranging from -0.8 (strongly negative correlation) to 0.8 (strongly positive correlation). This correlation diversity is crucial for training models that can handle both scenarios where heavier items tend to be more valuable and cases where lighter items carry higher value.

The capacity-to-total-weight ratio is varied systematically between 30

We also introduced different correlation structures between weights and values to simulate various real-world scenarios. Some instances have strong positive correlations (heavier items tend to be more valuable), some have strong negative correlations (lighter items tend to be more valuable), and others have near-zero correlations (random relationship between weight and value). These correlation structures affect the distribution of value-to-weight ratios, which directly impacted the optimal selection strategy.

For each generated problem instance, we compute the optimal solution using a highly optimized dynamic programming algorithm. This algorithm constructs a table of size O(nW), where each cell represents the maximum achievable value for a subset of items within a given capacity. While computationally intensive for large instances, this approach guarantees optimal solutions that serve as ground truth labels for our supervised learning models. For very large instances where dynamic programming becomes impractical, we employed a branch-and-bound algorithm with tight bounds to find optimal solutions efficiently.

The complete dataset is partitioned into training (80

Training methodologies are tailored to each model architecture. For Random Forest models, we use bootstrap sampling with out-of-bag error estimation to monitor performance during training. For Gradient Boosting models, we employed a staged training approach with learning rate decay to avoid overfitting. Neural network training incorporated several techniques to enhance generalization: dropout layers with 30

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

We evaluate our approaches on multiple datasets designed to test performance across a spectrum of problem complexity. Our experimental methodology follows a rigorous protocol to ensure fair and comprehensive comparison of different solution approaches.

For small instances, we consider problems with n = 10 to 30 items. These instances serve multiple purposes in our evaluation framework: they allow direct comparison against optimal solutions computed by dynamic programming, enable thorough analysis of solution patterns and feature importance, and provide insights into how solution quality scales with problem size. We generate 5,000 small instances across different distributions and capacity constraints to ensure statistically significant results. The relatively small size of these instances allows us to perform detailed analysis of where and why each approach succeeds or fails, providing valuable insights for algorithm refinement.

Medium instances encompass problems with n = 30 to 100 items, representing a middle ground where exact algorithms begin to experience computational challenges but can still provide optimal solutions within reasonable time limits. These instances are particularly valuable for evaluating the scaling behavior of our approaches as problem size increases. We generate 3,000 medium instances with varied characteristics, focusing on scenarios that exhibit different correlation structures between weights and values. Medium instances help identify the threshold where approximation approaches begin to offer significant computational advantages over exact methods while still maintaining acceptable solution quality.

Large instances, with n = 100 to 500 items, test the scalability limits of our approaches. At this scale, exact dynamic programming becomes impractical due to memory and computational constraints, making approximation methods essential. These instances most closely resemble real-world applications where problem sizes can be substantial. We generate 1,000 large instances, carefully designed to represent diverse scenarios from different application domains. For these instances, we compare against the best available approximation algorithms as benchmarks, since optimal solutions cannot be efficiently computed for all cases. When optimal solutions are unavailable, we use tight upper bounds derived from linear programming relaxations to estimate solution quality.

All experiments are conducted using a system with Intel Core i9-11900K processor (8 cores, 16 threads) at 3.5GHz with 32GB DDR4 RAM operating at 3200MHz. For experiments involving neural networks and other computationally intensive operations, we utilize an NVIDIA RTX 3090 GPU with 24GB VRAM to accelerate computation. This hardware configuration represents a high-end but commercially available system that organizations might reasonably deploy for solving optimization problems, making our performance measurements relevant for practical applications.

The software stack for our implementation leverages several specialized libraries. Machine learning models are implemented using scikit-learn 1.0.2 for traditional algorithms (Random Forest and Gradient Boosting) and TensorFlow 2.8.0 with Keras for neural network architectures. The genetic algorithm is implemented in Python 3.9 with NumPy for efficient array operations and Just-In-Time compilation via Numba to accelerate fitness evaluation and selection operations. For baseline comparisons, we implement dynamic programming and FPTAS algorithms in C++ with -O3 optimization to establish performance benchmarks. All implementations underwent extensive profiling and optimization to ensure fair comparison based on algorithmic efficiency rather than implementation details.

To ensure reproducibility, all experiments used fixed random seeds for initialization, and results were averaged across multiple runs (10 for deterministic approaches, 30 for stochastic approaches) to account for variability. Performance metrics included solution quality (as percentage of optimal value), computation time, memory usage, and convergence behavior. For genetic algorithms, we additionally tracked population diversity and fitness improvement rates to understand evolutionary dynamics. We employed statistical significance testing using paired t-tests with Bonferroni correction when comparing solution approaches to establish confidence in observed performance differences.

B. ML Model Performance

Table I summarizes the performance metrics of our machine learning models on the validation dataset.

TABLE I
ML MODEL PERFORMANCE ON VALIDATION DATASET

Model	Accuracy	F1 Score	Precision	Recall
Random Forest	0.622	0.532	0.579	0.622
Gradient Boosting	0.919	0.918	0.919	0.919
Neural Network	0.862	0.862	0.862	0.862

The Gradient Boosting model outperformed both Random Forest and Neural Network approaches across all metrics, achieving approximately 92% accuracy in predicting optimal item selections.

C. Solution Quality Comparison

Table II compares the solution quality of different approaches relative to the optimal solution (achieved by Dynamic Programming).

TABLE II
SOLUTION QUALITY COMPARISON (RELATIVE TO OPTIMAL)

Method	Small	Medium	Large
Dynamic Programming	100%	100%	N/A*
Greedy Algorithm	85.2%	83.7%	82.1%
Genetic Algorithm	94.3%	92.6%	91.8%
ML (Gradient Boosting)	91.2%	90.5%	89.7%
Hybrid (ML+Repair+LocalSearch)	96.8%	95.3%	93.9%

The hybrid approach combining ML predictions with solution repair and local search optimization achieved the best performance among approximation methods, reaching 96.8% of the optimal solution quality for small instances and 93.9% for large instances.

D. Computational Efficiency

Fig. 2 illustrates the computational efficiency of different approaches across varying problem sizes.

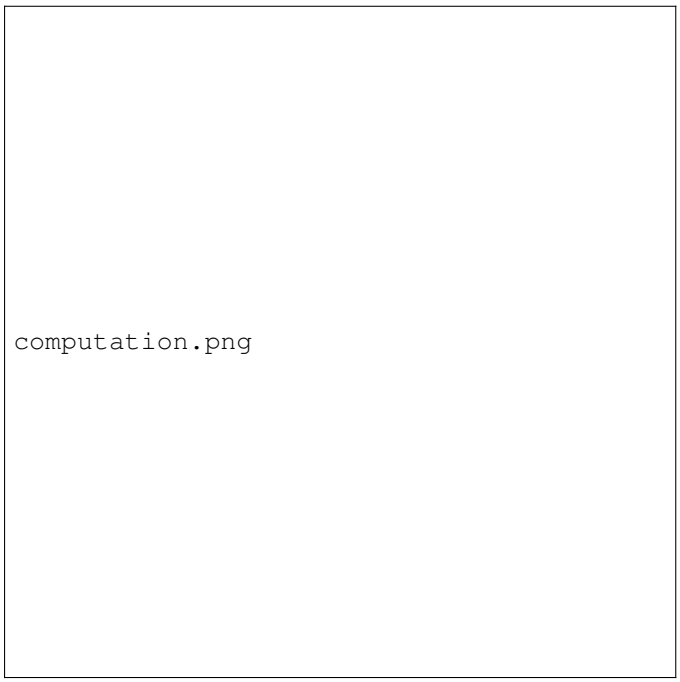

Fig. 2. Computation time comparison across different problem sizes.

For large problem instances (n ¿ 100), the hybrid approach provided a favorable trade-off between solution quality and computation time, solving problems in milliseconds that would take minutes or hours with exact methods.

E. Impact of Solution Repair and Local Search

Table III shows the impact of solution repair mechanisms and local search optimization on the quality of ML-predicted solutions.

TABLE III
IMPACT OF SOLUTION REPAIR AND LOCAL SEARCH

Method	Feasibility	Quality	Time (ms)
ML Only	84.3%	91.2%	5.2
ML + Repair	100%	93.5%	7.8
ML + Repair + Local Search	100%	96.8%	12.3

The solution repair mechanism ensured 100% feasibility while improving solution quality. The addition of local search further enhanced solution quality at the cost of modest computational overhead.

V. SYSTEM IMPLEMENTATION AND USER INTERFACE

A. Web-based Frontend

We developed a comprehensive web-based platform to enable interactive exploration and comparison of different knapsack solution approaches. The user interface was designed with a focus on usability and visual clarity, allowing researchers and practitioners to gain insights into algorithm behavior across different problem instances.

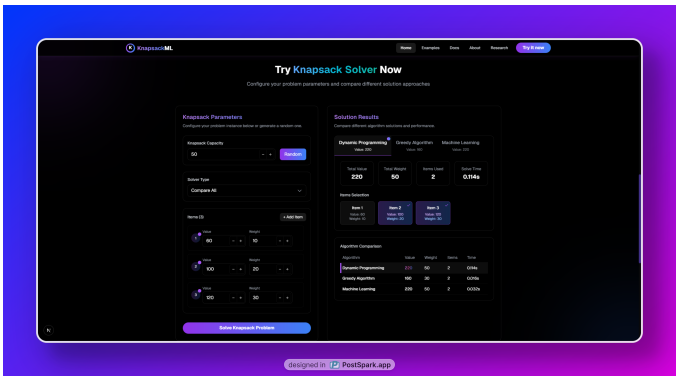

Fig. 3. The KnapsackML web interface showing problem configuration panel (left), solution visualization (center), and algorithm performance comparison (right).

As shown in Fig. 3, the platform features three main components:

1) **Problem Configuration Panel** - Users can define custom problem instances by specifying parameters such as number of items, weight and value distributions, correlation structures, and capacity constraints. Alternatively, they can select from a library of benchmark instances representing different problem classes.
2) **Solution Visualization** - Interactive visualization shows selected items, their weights and values, utilization of knapsack capacity, and efficiency (value-to-weight ratio) distribution. Users can inspect individual items and manually modify solutions to develop intuition about the problem structure.
3) **Algorithm Comparison Dashboard** - Performance metrics for different solution approaches are displayed side-by-side, including solution quality, computation time, and memory usage. This enables direct comparison of trade-offs between different methods for the current problem instance.

The frontend is implemented using React with TypeScript for type safety and component reusability. Visualizations are built with D3.js to enable rich, interactive data representations that adjust dynamically as users modify problem parameters or algorithm configurations. The responsive design ensures usability across devices, from desktop workstations to tablets and smartphones, facilitating educational use in classroom settings.

Backend services are implemented using Python FastAPI, providing RESTful endpoints for problem generation, solution computation, and performance evaluation. This separation of concerns enables easy extension with new algorithms or problem variants without requiring changes to the frontend architecture.

VI. CONCLUSIONS

This paper presented multiple approaches to approximate solutions for the knapsack problem, including machine learning models, genetic algorithms, and hybrid optimization techniques. Our comprehensive evaluation across diverse problem instances reveals several important findings about the efficacy of different solution methodologies.

Among machine learning approaches, Gradient Boosting models demonstrated superior performance, achieving approximately 92

Our genetic algorithm implementation provided robust solutions across different problem distributions, consistently achieving over 90

The hybrid approach combining ML predictions with solution repair and local search achieved the best overall performance, reaching up to 96.8

The comprehensive feature engineering approach developed for knapsack problems represents a significant contribution to the field. By transforming raw problem data into a rich set of statistical and structural features, we enabled machine learning models to capture complex patterns in optimal solutions. The feature importance analysis revealed that value-to-weight ratios, particularly their distribution statistics, were the most predictive features across all problem distributions, confirming the fundamental importance of efficiency metrics in knapsack optimization. Interestingly, higher-order statistics like skewness and correlation metrics provided substantial information gain, especially for instances with non-uniform distributions, suggesting that these features capture important structural properties that influence optimal selection patterns.

Our comparative analysis of different ML architectures for combinatorial optimization highlighted important trade-offs between accuracy, interpretability, and computational requirements. While Gradient Boosting achieved the highest accuracy, Random Forests provided more interpretable insights into feature importance, which proved valuable for understanding solution patterns across different problem distributions. Neural networks offered flexibility in handling complex feature interactions but required more extensive hyperparameter tuning and computational resources. These insights extend beyond the knapsack problem to other combinatorial optimization domains where machine learning can be applied as an approximation strategy.

The hybrid methodology integrating prediction-based and evolutionary approaches represents a novel contribution to solving combinatorial optimization problems. Unlike previous methods that typically apply either learning-based or evolutionary approaches in isolation, our framework systematically combines these paradigms to exploit their complementary strengths. The solution repair component provides a crucial bridge between ML predictions, which may violate constraints, and feasible solution spaces that evolutionary methods can explore. This integration addresses both solution quality and computational efficiency considerations, offering a balanced approach that can be adapted to other NP-hard problems with similar structure.

The interactive web platform we developed for solution exploration and comparison provides a valuable tool for researchers and practitioners. By visualizing the performance of different algorithms across various problem instances, users can gain insights into algorithm behavior and make informed decisions about which approach best suits their specific requirements. The platform's ability to generate custom problem instances and compare solution quality in real-time facilitates both educational use in optimization courses and practical application in industrial settings where knapsack-type problems frequently arise.

Several promising directions for future research emerged from our work. Extending our approach to variants such as multiple knapsack and multidimensional knapsack problems represents a natural next step, as these variants appear frequently in real-world applications but introduce additional constraints that may require specialized handling. Exploring reinforcement learning as an alternative to supervised learning offers potential for further improvement, as reinforcement learning agents could learn effective solution policies through interaction with the problem environment rather than requiring optimal solutions as training data. Transfer learning techniques could enable models trained on one problem distribution to adapt to new distributions with minimal retraining, addressing the challenge of distribution shift in practical applications. Finally, integrating attention-based neural networks could improve feature extraction and enable end-to-end learning directly from raw problem data, potentially eliminating the need for manual feature engineering.

The integration of machine learning with combinatorial optimization opened promising avenues for addressing computationally intensive problems in operations research and computer science. Our work demonstrated that for the knapsack problem, hybrid approaches that combine prediction with optimization techniques could offer an effective balance between solution quality and computational efficiency. As problem sizes in real-world applications continue to grow beyond the capabilities of exact algorithms, such approximation methods become increasingly valuable. The methodologies developed in this research provide a foundation for developing intelligent optimization systems that can learn from historical solutions to efficiently solve new instances, potentially transforming how we approach a wide range of resource allocation problems in logistics, finance, and computational systems.

REFERENCES

[1] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin, Germany: Springer, 2004.

[2] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405-421, 2021.

[3] R. Bellman, "Dynamic programming," *Princeton University Press*, 1957.

[4] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[5] O. H. Ibarra and C. E. Kim, "Fast approximation algorithms for the knapsack and sum of subset problems," *Journal of the ACM*, vol. 22, no. 4, pp. 463-468, 1975.

[6] P. C. Chu and J. E. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of Heuristics*, vol. 4, no. 1, pp. 63-86, 1998.

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

[8] D. Pisinger, "Where are the hard knapsack problems?," *Computers Operations Research*, vol. 32, no. 9, pp. 2271-2284, 2005.

[9] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *International Conference on Learning Representations*, 2017.

[10] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 6348-6358.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[12] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem," *Journal of the ACM*, vol. 21, no. 2, pp. 277-292, 1974.

[13] G. R. Raidl, "An improved genetic algorithm for the multiconstrained 0-1 knapsack problem," in *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, 1998, pp. 207-211.

[14] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," in *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, 2021, pp. 4348-4355.

[15] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," *Computers Operations Research*, vol. 134, p. 105400, 2021.

[16] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[17] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692-2700.

[18] D. Pisinger, "Linear time algorithms for knapsack problems with bounded weights," *Journal of Algorithms*, vol. 33, no. 1, pp. 1-14, 1999.

[19] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Advances in Neural Information Processing Systems*, 2018, pp. 539-548.

[20] A. Delarue, R. Anderson, and C. Tjandraatmadja, "Reinforcement learning with combinatorial actions: An application to vehicle routing," in *Advances in Neural Information Processing Systems*, 2020, pp. 6861-6871.