









Performance Analysis of Multiple Knapsack Problem Optimization Algorithms: A Comparative Study for Retail and SME Applications

Dewi Agustini Santoso^{1*}, Ifan Rizqa¹, Diana Aqmal¹, Farrikh Alzami¹, Nova Rijati¹, Aris Marjuni²

¹ Faculty of Computer Science, Universitas Dian Nuswantoro, Semarang 50131, Indonesia

² Faculty of Economic and Business, Universitas Dian Nuswantoro, Semarang 50131, Indonesia

Corresponding Author Email: dewi@dsn.dinus.ac.id

Copyright: ©2025 The authors. This article is published by IETA and is licensed under the CC BY 4.0 license (<http://creativecommons.org/licenses/by/4.0/>).

<https://doi.org/10.18280/isi.300224>

ABSTRACT

Received: 16 January 2025

Revised: 10 February 2025

Accepted: 14 February 2025

Available online: 27 February 2025

Keywords:

multiple knapsack problem, optimization algorithms, product bundling, parallel processing, dynamic programming, computational performance, metaheuristics, algorithm comparison

Multiple knapsack problem (MKP) optimization presents significant challenges in retail resource allocation, particularly for product bundling scenarios with dual constraints. While numerous algorithms exist for solving MKP, comprehensive comparisons focusing on real-world retail applications remain limited. This study evaluates seven distinct algorithms: Base Dynamic Programming (BDP), Numba-accelerated DP, parallel processing (P-P), parallel rolling (P-R), genetic algorithm (GA), greedy algorithm, and Branch & Bound. Using a real dataset of 88 Indonesian traditional products allocated across 7 knapsacks, we analyze algorithm performance through solution quality, execution time, memory utilization, and statistical validation. The algorithm evaluation employs a comprehensive TOPSIS-based multi-criteria decision framework with systematically allocated weights: solution quality (0.5), runtime efficiency (0.3), memory utilization (0.1), and resource optimization (0.1). These weights were determined through analytical hierarchy process considering practical implementation priorities in retail environments, where solution optimality takes precedence while maintaining computational efficiency. The TOPSIS analysis incorporates normalized performance metrics and validates ranking stability through sensitivity analysis, ensuring reliable algorithm recommendations. Results demonstrate that parallel rolling achieves optimal solutions (247.50 total value) with 99.66% resource utilization and 98.90% runtime improvement over traditional approaches. For resource-constrained environments, Branch & Bound offers 90% utilization with minimal computational overhead. Statistical analysis through TOPSIS confirms the superior performance of parallel variants (scores > 0.880) compared to basic implementations (scores < 0.800). Our findings provide evidence-based recommendations for algorithm selection based on business scale and computational resources, contributing to practical MKP implementation in retail product bundling scenarios.

1. INTRODUCTION

The multiple knapsack problem (MKP) represents a critical optimization paradigm in modern retail operations, particularly in the era of data-driven decision-making and resource optimization [1, 2]. As an NP-hard combinatorial optimization challenge, MKP extends beyond theoretical computer science into practical retail applications, where it addresses crucial business decisions in product bundling, inventory allocation, and resource distribution [3-5]. The complexity of contemporary retail operations, characterized by dual constraints of physical limitations (weight) and financial restrictions (budget), has elevated MKP from a theoretical construct to an essential tool for operational excellence [6, 7].

Recent developments in retail technology have highlighted the growing significance of efficient resource allocation, particularly in e-commerce and traditional retail bundling scenarios [6, 8]. The inherent complexity of MKP, which increases exponentially with the number of items and

knapsacks, necessitates sophisticated algorithmic solutions that balance solution quality with computational feasibility [9, 10]. This balance becomes particularly crucial in retail environments where decision-making often requires real-time optimization under multiple constraints.

Dynamic programming emerges as a foundational technique in this context, offering systematic approaches to decompose complex optimization challenges into manageable subproblems. In MKP applications, dynamic programming facilitates the methodical exploration of item combinations and their allocation across multiple knapsacks, creating a robust framework for solution optimization [11-13]. This approach has demonstrated particular efficacy in retail scenarios where solution accuracy directly impacts operational efficiency and profitability.

Schäfer and Zweers [14] stated that most research in MKP at the moment, did not consider each knapsack have own capacity restriction. Then, Babukarthik et al. [15] mentioned that the multi-constraint Knapsack problem (KP) remains the major challenge in weight and capacity to minimize energy

consumption. Then, Dell'Amico et al. [1] used smaller instances for MKP due to their computational complexity. Moreover, some researchers have addressed dual-constraint scenarios but lack comprehensive algorithmic comparison. For instance, the studies [16, 17] compared ACO and traditional ACO, and clustered orienteering problem families respectively.

While extensive research exists on MKP algorithms [16-20], three critical gaps persist in current literature: (1) Limited comparative analysis of algorithm performance under retail-specific dual constraints (weight and budget); (2) Insufficient evaluation of implementation requirements across different business scales (enterprise vs. SME); (3) Lack of comprehensive statistical validation for algorithm selection in practical retail applications. Recent algorithmic innovations, including parallel processing architectures, metaheuristic optimization, and hybrid methodologies, have shown promising computational efficiency improvements [21-26]. However, existing research predominantly focuses on either theoretical performance analysis or single-constraint implementations, creating a significant knowledge gap in understanding algorithmic behavior under dual-constraint scenarios common in retail applications.

This knowledge gap becomes particularly significant when considering the diverse computational infrastructure between large retail enterprises and Small-Medium Enterprises (SMEs), where resource availability drastically impacts algorithm selection and implementation strategies [11, 27, 28]. The practical implications of this gap are evidenced in recent studies: Zhao et al. [6] highlighted the challenges in optimizing case pack sizes in retail supply chains, while Gecili and Parikh [7] demonstrate the complexity of shelf space allocation under multiple constraints. These real-world applications underscore the need for a comprehensive understanding of algorithm performance under varying resource constraints and business scales.

For instance, the work [29] on cooperative content caching demonstrates the broader applicability of multi-constraint optimization, particularly in resource-limited environments. This is complemented research [2] on Modified Artificial Bee Colony algorithms, which, while innovative, primarily focuses on single-dimension performance metrics without comprehensive comparative analysis under dual constraints. Dell'Amico et al.'s [1] mathematical models provide valuable theoretical foundations, but leave open questions about practical implementation in retail scenarios. Similarly, Boukhari et al.'s [22] hybrid algorithm shows promise in solving MKP with setup costs, yet requires further validation in real-world retail applications.

This research makes several novel contributions to address these gaps: 1) Methodological advancement which consist of development of a comprehensive evaluation framework specifically designed for retail-oriented dual-constrained MKP, introduction of a hybrid statistical validation approach that combines traditional metrics with practical retail performance indicators, creation of implementation guidelines calibrated to different operational scales; 2) Practical applications, which consists of evidence-based algorithm selection criteria for retail scenarios, quantitative assessment of resource requirements for various implementation scales, performance benchmarks for different operational contexts; 3) Theoretical extensions which consist of integration of parallel processing techniques with traditional optimization approaches, enhanced understanding of algorithm behavior

under dual constraints, and Novel insights into the relationship between computational resources and solution quality.

The primary objectives of this study are to: 1) Evaluate and compare eight distinct algorithms for solving dual-constrained MKP in terms of solution quality, computational efficiency, and resource utilization; 2) Analyze algorithm scalability and performance stability through comprehensive statistical testing; 3) Provide evidence-based recommendations for algorithm selection based on business scale, computational resources, and optimization requirements; 4) Establish a framework for algorithm selection in retail product bundling optimization.

Our methodological approach combines rigorous algorithmic analysis with practical retail considerations. By evaluating seven distinct algorithms—BDP, Numba-accelerated DP, parallel processing, parallel rolling, genetic algorithm, greedy algorithm, and Branch & Bound—we provide a comprehensive performance analysis framework that addresses both theoretical efficiency and practical implementation requirements. The evaluation utilizes a carefully constructed dataset of 88 Indonesian traditional products allocated across 7 knapsacks, representing a realistic retail bundling scenario while maintaining mathematical tractability.

This study's significance extends beyond traditional algorithm comparison, making substantial contributions to both theory and practice, in matter of Theoretical Contributions, we present development of a comprehensive evaluation framework for dual-constrained MKP algorithms that bridges theoretical performance and practical implementation and establishment of implementation guidelines that consider varying operational scales and resource limitations. In matter of practical contributions, we present evidence-based algorithm selection criteria tailored to specific retail scenarios and business scales, detailed quantification of resource requirements enabling informed implementation planning, and clear performance expectations and benchmarks for different operational scales.

These contributions directly address the identified research gaps while providing actionable insights for both researchers and practitioners in the retail sector. The findings are particularly relevant for organizations navigating the balance between computational efficiency and practical implementation constraints.

The remainder of this paper is organized as follows: Section II presents the formal mathematical formulation and algorithmic implementations, incorporating both theoretical foundations and practical considerations. Section III details the experimental results and statistical analysis, providing comprehensive performance comparisons across multiple metrics. Section IV discusses implementation implications and provides specific recommendations for different business scales and computational environments. Finally, Section V concludes the study and suggests promising directions for future research.

2. MATERIAL AND METHODS

2.1 Problem formulation and mathematical model

The MKP with dual constraints presents a combinatorial optimization challenge in which n heterogeneous items must be optimally allocated across m distinct knapsacks [30]. This

study addresses a practical retail bundling scenario with the following formal mathematical representation:

Given:

- Set of n items $I = \{1, \dots, n\}$, where each item $i \in I$ has: Value (v_i): representing item utility/desirability; Weight (w_i): physical weight in kilograms; Price (p_i): monetary cost in IDR
- Set of m knapsacks $K = \{1, \dots, m\}$, where each knapsack $j \in K$ has: Weight capacity (C_j): maximum allowable total weight; Budget constraint (B_j): maximum allowable total cost; Unique size-cost characteristics reflecting real retail bundles

The experimental dataset was systematically constructed to represent realistic retail bundling scenarios while ensuring mathematical tractability. The study incorporates 88 traditional Indonesian products strategically segmented into two primary categories: beverages ($n=45$) and snacks ($n=43$), reflecting typical product mix ratios in traditional retail operations. For each product, three key metrics were carefully calibrated: value coefficients ranging from 1.0 to 10.0 units derived from normalized customer preference data, weight parameters between 0.11 and 1.0 kilograms aligned with standard retail packaging constraints, and price variables from 6,000 to 60,000 IDR mapped to actual market price distributions.

The allocation framework comprises seven knapsacks, designed to represent distinct retail bundling strategies across multiple market segments. These configurations include premium bundles (4.0 kg, 250,000 IDR; 3.0 kg, 300,000 IDR), standard bundles (3.0 kg with budgets of 180,000-250,000 IDR), and value bundles (2.0 kg, 150,000 IDR; 1.5 kg, 160,000 IDR). This empirically derived configuration from market analysis of Indonesian retail operations balances operational constraints with customer segment requirements. The knapsack capacities and budget constraints were specifically calibrated to reflect realistic handling limitations in retail environments, accommodate diverse market segments' purchasing power, enable meaningful algorithmic performance comparison, and maintain computational feasibility for systematic analysis.

Mathematical model:

$$\text{Maximize: } \sum_{j=1}^m \sum_{i=1}^n v_i x_{ij} \quad (1)$$

Subject to:

1. Weight Constraint:

$$\sum_{i=1}^n w_i x_{ij} \leq C_j \forall j \in \{1, \dots, m\} \quad (2)$$

2. Budget Constraints:

$$\sum_{i=1}^n p_i x_{ij} \leq B_j \forall j \in \{1, \dots, m\} \quad (3)$$

3. Item Uniqueness:

$$\sum_{j=1}^m x_{ij} \leq 1 \forall i \in \{1, \dots, n\} \quad (4)$$

4. Binary Decision Variables:

$$x_{ij} \in \{0,1\} \forall i,j \quad (5)$$

where, x_{ij} is binary decision variable (1 if item i is assigned to knapsack j , 0 otherwise); v_i represents the value of item i ; w_i represents the weight of item i ; p_i represents the price of item i ; C_j represents the weight capacity of knapsack j ; B_j represents the budget capacity of knapsack j .

2.2 Implementation

This section details eight algorithm implementations for solving the dual-constrained MKP, ranging from exact methods to heuristic approaches. Each implementation is analyzed for theoretical complexity, memory requirements, and optimization strategies.

2.2.1 BDP

BDP enhances the traditional approach through numpy vectorization and improved memory management. This implementation maintains optimality while introducing several key optimizations for computational efficiency.

1. Core Architectural Improvements

1) Numpy-based array operations for vectorized computation; 2) Efficient memory allocation through data type optimization; 3) Decimal precision handling through integer scaling; 4) Enhanced state transition management

2. Key Implementation Features can be seen in Algorithm 1

Algorithm 1: Snippet python code for BDP

```

1  def dp_solve(self, n, m, weights, prices, values,
2     capacities, budgets):
3     max_capacity = int(max(capacities) * 100)
4     dp = np.zeros((n + 1, m, max_capacity + 1,
5     max_budget + 1), dtype=np.float32)
6     for i in range(1, n + 1):
7         for j in range(m):
8             for w in range(int(capacities[j] * 100) + 1):
9                 for b in range(int(budgets[j] + 1):
10                    if int(weights[i-1] * 100) <= w and
11                       prices[i-1] <= b:
12                        dp[i, j, w, b] = max(dp[i-1, j, w, b],
13                        dp[i-1, j, w - int(weights[i-1] * 100),
14                        int(b - prices[i-1])] + values[i-1])
15                    else:
16                        dp[i, j, w, b] = dp[i-1, j, w, b]
17     return dp

```

3. Technical Optimizations in Space Complexity

The technical optimizations in BDP implementation focus on two critical areas: memory management and computational enhancements. Memory management is implemented through several sophisticated strategies that optimize resource utilization. The implementation uses np.float32 instead of the default float64 data type to reduce memory footprint while maintaining numerical precision. Pre-allocated arrays with optimal data types are employed to minimize memory allocation overhead during computation. Integer-based computations are implemented to improve numerical accuracy and reduce floating-point errors, while efficient memory access patterns optimize data retrieval and storage operations.

Computational enhancements are achieved through several key mechanisms: vectorized operations for state updates significantly improve processing speed by performing operations on entire arrays simultaneously, integer scaling techniques ensure decimal precision while maintaining computational efficiency, improved array indexing strategies optimize data access patterns, and better memory locality exploitation enhances cache utilization and reduces memory access latency. These technical optimizations able to create an efficient implementation that balances memory usage with computational performance while maintaining solution accuracy.

In advanced matter, we could employ employs a Rolling Array technique, significantly reducing space complexity from $O(n \times m \times W \times B)$ to $O(2 \times m \times W \times B)$ by adopting a two-array approach instead of maintaining $n+1$ arrays. This implementation enables more efficient memory utilization without compromising computational accuracy in the optimization process. Then, adopt bit-level state representation using `np.int8` for decision matrices, replacing the more memory-intensive `float32` implementation. This approach achieves a 75% reduction in per-cell memory requirements while maintaining efficient state tracking with minimal overhead. The implementation is reinforced with memory management mechanisms involving strategic garbage collection post-knapsack processing and unused state clearing, ensuring peak memory utilization remains below $O(2 \times m \times W \times B)$. As worth mentioning, the subsection 2.2.4 are explanation using the optimized DP using parallel rolling.

2.2.2 Numba-accelerated DP

Numba-accelerated DP extends the BDP approach by leveraging Just-In-Time (JIT) compilation for machine-level optimization. This implementation achieves significant performance improvements while maintaining solution optimality.

1. Core Implementation Architecture as in Algorithm 2

Algorithm 2: Snippet code for Numbas-accelerated DP

```

1  @staticmethod
2  @njit
3  def dp_solve(n, m, weights, prices, values, capacities,
4             budgets):
5      max_capacity = int(max(capacities) * 100)
6      max_budget = int(max(budgets))
7      dp = np.zeros((n + 1, m, max_capacity + 1,
8                   max_budget + 1), dtype=np.float32)
9      for i in range(1, n + 1):
10         for j in range(m):
11             for w in range(int(capacities[j] * 100) + 1):
12                 for b in range(int(budgets[j]) + 1):
13                     if int(weights[i-1] * 100) <= w and
14                        prices[i-1] <= b:
15                         dp[i, j, w, b] = max(dp[i-1, j, w, b],
16                                             dp[i-1, j, w - int(weights[i-1] *
17                                             100), int(b - prices[i-1])] +
18                                             values[i-1])
19                     else:
20                         dp[i, j, w, b] = dp[i-1, j, w, b]
21
22     return dp

```

2. Numba Optimization Features

The Numba optimization features leverage advanced JIT compilation strategies and robust performance optimization

techniques to enhance computational efficiency. JIT compilation strategies are implemented through sophisticated function specialization for data types, enabling optimized machine code generation. The implementation incorporates loop optimization and unrolling techniques that maximize execution efficiency, while SIMD instruction generation capabilities harness modern processor architectures for parallel computation. Cache-aware memory access patterns are implemented to minimize memory latency and optimize data retrieval. Performance optimizations are achieved through several key mechanisms: machine code generation for core computations significantly reduces execution overhead, while the elimination of Python interpreter overhead enhances raw computational speed. The implementation also employs automatic vectorization of operations to leverage hardware-specific capabilities, complemented by hardware-specific optimizations that take advantage of available system resources. These optimization features work in concert to create a highly efficient implementation that significantly outperforms traditional Python execution while maintaining solution accuracy and reliability.

3. Technical Implementation Details

The technical implementation details of the Numba-accelerated DP focus on two critical aspects: memory management and computational enhancements. The memory management system implements contiguous array allocation strategies to optimize memory access patterns and enhance cache utilization. This is supported by type-specific memory layouts that maximize memory efficiency for different data types, complemented by optimized cache utilization mechanisms that reduce memory access latency. The implementation also features reduced memory access latency through careful memory organization and access pattern optimization. On the computational side, the implementation leverages compiled loop execution capabilities that significantly enhance processing speed. This is supported by optimized boundary checking mechanisms that minimize computational overhead while maintaining solution validity. The system employs efficient register allocation strategies to maximize processor utilization, while reduced function call overhead minimizes computational bottlenecks. These technical features are carefully integrated to create a highly optimized implementation that balances memory efficiency with computational performance.

4. Key Improvements over BDP by Implementation Considerations

We need consider for deployment requirement, such as LLVM compiler infrastructure, CPU with SIMD support, Sufficient memory capacity, Proper environment setup. Also in matter of Optimization Guidelines can follows Function signature type specification, Array contiguity maintenance, Compiler directive optimization, Memory access pattern design.

2.2.3 Parallel processing implementation

Parallel processing enhances the Numba-accelerated approach by introducing multi-threaded computation and parallel array operations. This implementation leverages both thread-level parallelism and SIMD operations for maximum performance.

1. Parallel Architecture Configuration as in Algorithm 3

Algorithm 3: Snippet code for parallel processing with Numba

```

1  parallel.set_num_threads(4)
2  @staticmethod
3  @njit(parallel=True)
4  def dp_solve(n, m, weights, prices, values, capacities,
5             budgets):
6      max_capacity = int(max(capacities) * 100)
7      max_budget = int(max(budgets))
8      dp = np.zeros((n + 1, m, max_capacity + 1,
9                    max_budget + 1), dtype=np.float32)
10     for i in prange(1, n + 1):
11         for j in range(m):
12             for w in range(int(capacities[j] * 100) + 1):
13                 for b in range(int(budgets[j]) + 1):
14                     if int(weights[i-1] * 100) <= w and
15                        prices[i-1] <= b:
16                         dp[i, j, w, b] = max(dp[i-1, j, w, b],
17                                             dp[i-1, j, w - int(weights[i-1] *
18                                             100), int(b - prices[i-1])] +
19                                             values[i-1])
16     else:
17         dp[i, j, w, b] = dp[i-1, j, w, b]
18     return dp

```

2. Parallelization Strategy

The parallel processing implementation employs a sophisticated multi-level task distribution framework designed to optimize computational efficiency while maintaining solution integrity. Our approach incorporates three primary optimization layers:

Layer 1: Thread-Level Parallelization. The implementation utilizes Numba's parallel processing capabilities through the `@njit(parallel=True)` decorator, enabling efficient thread management (line code 2-9). This structure facilitates parallel item processing while maintaining sequential knapsack evaluation to prevent resource conflicts. Empirical analysis demonstrates a significant reduction in processing time from 79.16 seconds (base implementation) to 1.21 seconds, achieving an 80.47x speedup (more explanation in Results Section)

Layer 2: Resource Distribution Optimization. Our implementation incorporates dynamic workload balancing through strategic task partitioning:

- Computation Distribution, which Parallel item evaluation across available threads, Synchronized state updates for solution consistency, Memory-aware task allocation minimizing thread contention
- Memory Access Optimization, which Thread-local storage for intermediate computations, Cache-aligned data structures reducing memory latency, Optimized memory access patterns enhancing cache utilization.
- Load Balancing Mechanism can be seen in line code 1 and 8.

Layer 3: Synchronization Control. The implementation maintains solution consistency through carefully designed synchronization mechanisms as State Management: Atomic updates for shared resource modifications, Barrier synchronization at critical computation points, and Efficient thread coordination minimizing overhead.

3. Technical Implementation Features

The technical implementation features encompass two key areas of focus: parallel processing components and memory architecture. The parallel processing components are built around a sophisticated thread pool management system that efficiently handles task distribution across available

processors. This is supported by a work queue distribution mechanism that optimizes task allocation and processing flow. The implementation incorporates essential synchronization primitives to maintain data consistency and prevent race conditions during parallel execution. To ensure robust concurrent processing, comprehensive race condition prevention mechanisms are implemented throughout the system. The memory architecture is designed with careful consideration of thread-local storage capabilities to minimize contention and maximize parallel efficiency. This is complemented by shared memory management strategies that optimize resource utilization across threads. The implementation also employs cache line optimization techniques to enhance memory access performance, supported by memory fence implementation that ensures proper synchronization of memory operations across multiple threads. These technical features work in concert to create a highly efficient parallel processing system that maintains both performance and reliability.

4. Key Improvements over Numba DP

- a. Implementation Benefits: Scalable with CPU cores, better resource distribution, Improved cache utilization, Reduced memory pressure.
- b. Trade-off: Increased implementation complexity, Hardware dependencies, Thread coordination overhead, Non-linear scaling beyond 4 cores, System resource requirements.

5. Implementation Requirements

- a. Hardware Prerequisites: Multi-core processor, Sufficient L3 cache, Adequate memory bandwidth, NUMA architecture awareness.
- b. Software Dependencies: Threading library support, Numba parallel features, Memory management tools, Synchronization primitives.

6. Optimization Guidelines

- a. Thread Management: Optimal thread count selection, Work distribution strategy, Synchronization minimization, Cache coherency maintenance.
- b. Memory Management: Thread-local allocation, Cache-line alignment, Memory access patterns, False sharing prevention.

The parallel implementation offers improved resource utilization and memory efficiency compared to Numba-only implementation, making it particularly suitable for large-scale retail applications with multiple CPU cores available. The slight increase in runtime is offset by better scalability and resource management.

2.2.4 Parallel rolling

Parallel rolling combines multi-threaded computation with memory optimization through state space rolling. This implementation achieves memory efficiency while maintaining parallel processing benefits through a three-state matrix rotation technique.

1. Core Implementation Architecture as seen in Algorithm 4

Algorithm 4: Parallel rolling

```

1  @staticmethod
2  @njit(parallel=True)
3  def dp_solve(n, m, weights, prices, values, capacities,
4             budgets):
5      max_capacity = int(max(capacities) * 100)
6      max_budget = int(max(budgets))
7      # Three-state matrix instead of n+1 states

```

```

7  dp = np.zeros((3, m, max_capacity + 1,
8  max_budget + 1), dtype=np.float32)
9  decisions = np.zeros((n, m, max_capacity + 1,
10 max_budget + 1), dtype=np.int8)
11 for i in range(1, n + 1):
12     curr = i % 3 # Current state
13     prev = (i - 1) % 3 # Previous state
14     for j in prange(m):
15         for w in range(int(capacities[j] * 100) + 1):
16             for b in range(int(budgets[j]) + 1):
17                 if int(weights[i-1] * 100) <= w and
18                    prices[i-1] <= b:
19                     if dp[prev, j, w - int(weights[i-1] *
20                        100), int(b - prices[i-1])] +
21                        values[i-1] > dp[prev, j, w, b]:
22                         dp[curr, j, w, b] = dp[prev, j, w,
23                            b] - int(weights[i-1] * 100), int(b -
24                            prices[i-1])] + values[i-1]
25                         decisions[i-1, j, w, b] = 1
26             else:
27                 dp[curr, j, w, b] = dp[prev, j, w,
28                    b]
29     return dp[(n-1) % 3], decisions

```

2. Memory Optimization Strategy

The memory optimization strategy in parallel rolling implementation centers around two critical components: state space rolling and decision tracking. The state space rolling mechanism employs an innovative three-state matrix rotation approach that significantly reduces memory footprint while maintaining computational efficiency. This system implements current/previous state tracking to manage state transitions effectively, coupled with efficient state transition mechanisms that optimize memory usage during computation. The implementation maintains a minimal memory footprint through careful resource management, ensuring optimal performance with reduced memory requirements. Complementing this, the decision tracking component utilizes a compact decision matrix design that efficiently stores solution choices. This is supported by efficient backtracking support mechanisms that enable solution reconstruction when needed, while employing a binary state representation to minimize memory usage. The implementation also features memory-efficient solution reconstruction capabilities that allow for detailed solution recovery without significant memory overhead. These optimization strategies work together to create a memory-efficient implementation that maintains high performance while significantly reducing memory requirements compared to traditional approaches.

3. Parallel Processing Features

The parallel processing features of the parallel rolling implementation are structured around two fundamental components: thread management and memory architecture. The thread management system implements task parallelization across knapsacks through an efficient distribution mechanism that optimizes workload allocation. This is supported by thread-safe state updates that ensure data consistency during parallel execution, complemented by synchronized memory access controls that prevent data corruption. The implementation employs load-balanced work distribution strategies to ensure optimal resource utilization across all available processors. The memory architecture is specifically designed to support parallel processing efficiency, implementing cache-friendly state rotation mechanisms that

optimize memory access patterns. The system utilizes thread-local computation capabilities to minimize contention between threads, supported by optimized memory access patterns that enhance overall performance. These features are further strengthened by reduced memory contention mechanisms that ensure smooth parallel execution. Together, these parallel processing features create a highly efficient implementation that maximizes computational throughput while maintaining data consistency and solution accuracy.

4. Key Improvements

- a. Performance Balance: Maintained parallel speedup, Reduced memory pressure, Improved cache performance, Efficient state management.
 - b. Trade-offs: Additional implementation complexity, State rotation overhead, Decision tracking overhead, Complex solution reconstruction, Synchronization requirements.
- ## 5. Implementation Considerations
- a. Technical Requirements: Multi-core processor support, Cache-coherent architecture, Memory management capabilities, Thread synchronization support.
 - b. Optimization Guidelines: State rotation timing, Memory access patterns, Thread work distribution, Cache line alignment.

The parallel rolling implementation represents an optimal balance between computational efficiency and memory usage, making it particularly suitable for memory-constrained environments while maintaining parallel processing benefits.

2.2.5 Genetic algorithm

The genetic algorithm provides a metaheuristic approach to MKP through evolutionary computation [3, 9, 11]. This implementation balances exploration and exploitation while handling dual constraints effectively.

1. Core Genetic Components as seen in Algorithm 5

Algorithm 5: Snippet code for genetic algorithm

```

1  @dataclass
2  class Individual:
3      chromosome: List[int] # Each gene represents
4      fitness: float = 0.0
5      is_feasible: bool = True
6  class MultipleKnapsackGA:
7      def __init__(self, population_size=100,
8                  generations=100, mutation_rate=0.1,
9                  elite_size=10, tournament_size=5):
10         self.population_size = population_size
11         self.generations = generations
12         self.mutation_rate = mutation_rate
13         self.elite_size = elite_size
14         self.tournament_size = tournament_size
15
16     def evaluate_fitness(self, position: np.ndarray) ->
17         float:
18         total_value = 0
19         for j in range(self.m):
20             weight_sum = sum(position[i, j] *
21                             self.items[i].weight for i in range(self.n))
22             price_sum = sum(position[i, j] *
23                             self.items[i].price for i in range(self.n))
24             if (weight_sum > self.knapsacks[j].capacity
25                 or price_sum > self.knapsacks[j].budget):
26                 return -np.inf

```

21	total_value += sum(position[i, j] * self.items[i].value
22	return total_value

2. Algorithmic Parameters

As seen in Table 1, The GA implementation utilizes carefully tuned parameters to balance exploration and exploitation. Population size of 150 individuals was chosen to maintain sufficient genetic diversity while remaining computationally manageable. The algorithm runs for 100 generations to allow adequate evolution time while preventing excessive computational overhead. A moderate mutation rate of 0.1 enables sufficient genetic variation without disrupting beneficial gene combinations. Elite size of 10 preserves the best solutions across generations while maintaining population diversity. Tournament size of 5 provides appropriate selection pressure - large enough to favor fitter individuals but small enough to prevent premature convergence.

Table 1. Genetic algorithm parameters

GA Parameter Settings	Values
Population Size	50,100,150,200
Number of Generations	100
Mutation Rate	0.1
Elite Size	10
Tournament Size	5

To validate the GA parameter settings, here we focus on population size. The result can be seen in Table 2.

Table 2. Population size analysis

Population Size	Total Value	Time (s)	Memory (MB)	Price Utilization (%)	Weight Utilization (%)
50	232	0.7260	145.09	93.42	79.95
100	230	1.5822	132.45	92.62	83.90
150	232.5	2.4622	145.35	93.62	76.72
200	232	3.4238	135.25	93.42	80.72

Based on comprehensive parameter sensitivity analysis on Table 2, comparing population sizes (50, 100, 150, 200) in the genetic algorithm implementation for MKP, population size 100 emerges as the optimal configuration. This conclusion is substantiated through rigorous empirical evaluation across multiple performance metrics: solution quality (total value = 230.0), computational efficiency (execution time = 1.5822s), resource utilization (memory = 132.45 MB), and optimization effectiveness (weight utilization = 83.90%, price utilization = 92.62%). While population size 150 achieved marginally higher solution quality (232.5), the minimal improvement of 1.09% does not justify the substantial increases in computational overhead (55.6% longer execution time) and memory consumption (9.74% higher), thus establishing population size 100 as the most efficient parameter setting for balancing solution quality and computational resources in this specific multiple knapsack optimization context.

As seen in Table 3, the evolution process is governed by four key control mechanisms. Selection Pressure utilizes an adaptive approach that automatically adjusts based on population diversity and convergence trends. Diversity Maintenance actively monitors and maintains genetic variation through strategic mutation and crossover operations. Convergence Control implements dynamic adjustments to

prevent premature convergence while ensuring efficient solution space exploration. Population Renewal mechanism introduces new genetic material by replacing 20% of the population with fresh individuals when diversity drops below critical thresholds, helping escape local optima while preserving good solutions.

Table 3. Genetic algorithm evolution controls

Evolution Controls	Value
Selection Pressure	Adaptive
Diversity Maintenance	Active
Convergence Control	Dynamic
Population Renewal	20%

3. Genetic Operators

Based on Table 4, the GA implementation incorporates three essential genetic operators, each carefully designed to ensure effective solution space exploration and exploitation:

- 1) Selection Mechanism employs a multi-faceted approach combining tournament selection for parent choice, elite preservation to maintain best solutions, fitness-based ranking to guide selection pressure, and diversity-aware selection to prevent premature convergence. Tournament selection with size 5 provides balanced selection pressure, while elitism preserves the top 10 solutions across generations.
- 2) Crossover Strategy utilizes a sophisticated approach with four key components. Two-Point Crossover enables effective genetic material exchange between parents. Constraint-Aware Recombination ensures offspring validity by respecting knapsack capacity and budget constraints. Solution Repair mechanisms correct any constraint violations post-crossover. Feasibility Preservation maintains solution validity throughout the evolutionary process.
- 3) Mutation Operations implement four complementary strategies. Random Gene Modification allows for exploration of new solution spaces. Intelligent Mutation applies problem-specific knowledge to guide modifications. Adaptive Rate Adjustment dynamically modifies mutation probability based on population diversity. Constraint Satisfaction ensures all mutations maintain solution feasibility within knapsack constraints.

Table 4. Genetic operators

Selection Mechanism	Crossover Strategy	Mutation Operations
- Tournament Selection	- Two-Point Crossover	- Random Gene Modification
- Elite Preservation	- Constraint Aware Recombination	- Intelligent Mutation
- Fitness-Based Ranking	- Solution Repair	- Adaptive Rate Adjustment
- Diversity-Aware Selection	- Feasibility Preservation	- Constraint Satisfaction

These genetic operators work synergistically to achieve the observed performance metrics, including solution quality of 230.0 and resource utilization of 92.62% for price and 83.90% for weight capacity. The operators' design particularly emphasizes maintaining solution feasibility while enabling effective search space exploration.

4. Implementation Enhancements

The implementation enhancements in the genetic algorithm

focus on two crucial areas: constraint handling and performance optimization. In constraint handling, the implementation employs a sophisticated repair mechanism that corrects infeasible solutions while maintaining genetic diversity. This is complemented by carefully designed penalty functions that guide the search toward feasible regions of the solution space. The system implements robust feasibility preservation mechanisms to maintain solution validity throughout the evolutionary process, supported by comprehensive solution validation procedures that ensure all constraints are satisfied. Performance optimization is achieved through several key mechanisms: parallel fitness evaluation capabilities that leverage multiple processors for increased computational efficiency, sophisticated caching mechanisms that reduce redundant calculations, and early stopping criteria that prevent unnecessary computational overhead. The implementation also features advanced population management techniques that maintain genetic diversity while focusing the search on promising regions of the solution space. These enhancement features work synergistically to create an efficient and reliable genetic algorithm implementation that effectively balances solution quality with computational performance while maintaining strict adherence to problem constraints.

5. Implementation Trade-offs

- a. Advantages: Low Memory Requirements, Anytime Solution Availability, Population Diversity, Parallel Potential.
- b. Limitations: Non-Guaranteed Optimality, Parameter Sensitivity, Convergence Variance, Solution Variability.

6. Optimization Guidelines:

- a. Parameter Tuning: Population Size Selection, Generation Count Optimization, Mutation Rate Adjustment, Tournament Size Calibration.
- b. Implementation Focus: Constraint Satisfaction, Diversity Maintenance, Convergence Control, Solution Quality Balance.

The genetic algorithm implementation provides a robust alternative when optimal solutions aren't strictly required, offering good solution quality with moderate computational resources [30, 31]. This makes it particularly suitable for larger problem instances where exact methods become computationally prohibitive.

2.2.6 Greedy algorithm

The greedy algorithm provides a fast, deterministic heuristic approach by making locally optimal choices based on value density metrics [32, 33]. This implementation achieves efficient resource utilization through strategic item selection.

1. Core Implementation Architecture as in Algorithm 6

Algorithm 6: Snippet code for greedy algorithm

```

1 class GreedyCriteria(Enum):
2     VALUE = "value"
3     VALUE_PER_WEIGHT = "value_per_weight"
4     VALUE_PER_PRICE = "value_per_price"
5     VALUE_PER_RESOURCE = "value_per_resource"
6
7 @dataclass
8 class Item:
9     def value_per_resource(self) -> float:
10        return self.value / (self.weight + self.price) if

```

```

        (self.weight + self.price) > 0 else float('inf')
11
12 class MultipleKnapsackGreedy:
13     def solve(self):
14         sorted_knapsacks = sorted(self.knapsacks,
15                                   key=lambda k: k.capacity/k.budget if
16                                   k.budget > 0 else float('inf'), reverse=True)
17         available_items = self.items.copy()
18         results = []
19
20         for knapsack in sorted_knapsacks:
21             sorted_items = self.sort_items(available_items)
22             selected_items = []
23             remaining_items = []
24             total_value = 0.0
25             # Greedy selection process
26             for item in sorted_items:
27                 if knapsack.can_add_item(item):
28                     if knapsack.add_item(item):
29                         selected_items.append(item)
30                         total_value += item.value
31                     Else:
32                         remaining_items.append(item)
33             available_items = remaining_items

```

2. Selection Strategy

- a. Value Density Metrics
- b. Sorting Criteria: Knapsack Efficiency Ratio, Item Value Density, Resource Consumption Rate, Combined Utility Measure

3. Implementation Features

- a. Core Components: Single-pass item selection, Greedy choice function, Constraint validation, Resource tracking
- b. Optimization Elements: Efficient sorting, Early termination, Resource monitoring, Solution construction

4. Implementation Advantages

- a. Computational Benefits: Minimal runtime overhead, Linear memory scaling, Deterministic behavior, Simple implementation
- b. Practical Benefits: No parameter tuning, Immediate solutions, Predictable performance, Easy maintenance

5. Solution Characteristics

- a. Resource Utilization: Memory Efficiency: high; CPU Utilization: minimal; I/O Requirements: negligible; Storage Needs: constant

6. Implementation Trade-offs

- a. Advantages: Extremely fast execution, Simple implementation, Deterministic results, Low resource requirements
- b. Limitations: Sub-optimal solutions, Local decision making, No solution refinement, Fixed selection criteria

7. Key Improvements over Previous Approaches

- a. Performance Gains: Fastest execution time, Minimal memory overhead, Simple computation model, Immediate results.
- b. Resource Efficiency: Constant memory usage, Single-pass processing, No iteration required, Linear scaling

8. Implementation Guidelines

- a. Selection Criteria: Value density metric choice, Sorting strategy selection, Constraint handling, Resource

balancing

- b. Optimization Focus: Sort efficiency, Memory management, Constraint checking, Solution construction

The Greedy approach provides an extremely fast solution with reasonable quality, making it particularly suitable for real-time applications or when computational resources are severely constrained. Its performance characteristics make it an excellent choice for initial solution generation or when quick approximations are acceptable.

2.2.7 Branch & Bound algorithm

Branch & Bound provides a systematic approach to finding optimal or near-optimal solutions through intelligent search space exploration and pruning [3, 29, 34]. This implementation utilizes priority-based searching with efficient bounding mechanisms.

1. Core Data Structures as in Algorithm 7

Algorithm 7: Snippet code for Branch & Bound

```
1  @dataclass(order=True)
2  class PrioritizedNode:
3      priority: float
4      level: int = field(compare=False)
5      value: float = field(compare=False)
6      weight: Dict[int, float] = field(compare=False)
7      price: Dict[int, float] = field(compare=False)
8      assigned_items: Dict[int, List[Item]] =
9      field(compare=False)
10     bound: float = field(compare=False)
11
12 class MultipleBranchAndBoundSolver:
13     def solve(self):
14         pq = PriorityQueue()
15         # Initialize root node
16         initial_weights = {i: k.capacity for i, k in
17             enumerate(self.knapsacks)}
18         initial_budgets = {i: k.budget for i, k in
19             enumerate(self.knapsacks)}
20         root_bound = self.calculate_bound(0, 0,
21             initial_weights, initial_budgets)
22         root_node = PrioritizedNode(
23             priority=-root_bound, # Negative for max-
24             heap behavior
25             level=0,
26             value=0,
27             weight=initial_weights,
28             price=initial_budgets,
29             assigned_items={i: [] for i in range(self.m)},
30             bound=root_bound)
31         pq.put(root_node)
```

2. Bounding Strategy

Algorithm 8: Snippet code for calculate bound

```
1  def calculate_bound(self, level: int, curr_value: float,
2     remaining_weights: Dict[int, float],
3     remaining_budgets: Dict[int, float]) -> float:
4
5     bound = curr_value
6     remaining_items = self.items[level:]
7
8     for item in remaining_items:
9         min_fraction = 1.0
10        selected_knapsack = -1
```

```
8     for k_id, remaining_weight in
9     remaining_weights.items():
10        if remaining_weight >= item.weight and
11        remaining_budgets[k_id] >= item.price:
12            fraction = min(remaining_weight /
13                item.weight, remaining_budgets[k_id] /
14                item.price, 1.0)
15            if fraction > min_fraction:
16                min_fraction = fraction
17            selected_knapsack = k_id
```

3. Search Space Management

The search space management in the Branch & Bound implementation is structured around two key components: node exploration strategy and pruning mechanisms. The node exploration strategy employs a best-first search approach using a priority queue system to efficiently navigate the solution space. This is supported by depth-based exploration control that manages the search depth to balance between exploration and exploitation. The implementation features efficient node pruning techniques that eliminate unpromising branches of the search tree, complemented by solution space partitioning strategies that effectively divide the problem into manageable subproblems. The pruning mechanisms are implemented through several sophisticated approaches: upper bound comparison techniques that quickly identify and eliminate suboptimal branches, rigorous feasibility checking procedures that maintain solution validity, comprehensive resource constraint validation that ensures all solutions meet problem constraints, and dominated solution elimination strategies that remove redundant search paths. These search space management features work together to create an efficient implementation that effectively explores the solution space while minimizing computational overhead through strategic pruning and exploration control.

4. Implementation Features

The implementation features of the Branch & Bound algorithm encompass two primary aspects: search optimization and memory management. The search optimization process is built around priority-based exploration that efficiently guides the search toward promising regions of the solution space. This is enhanced by efficient bound computation mechanisms that quickly evaluate the potential of each branch, complemented by early termination capabilities that prevent unnecessary exploration of unpromising paths. The system maintains comprehensive solution tracking procedures that record the best solutions found during the search process. Memory management is implemented through several sophisticated strategies: compact node representation techniques minimize memory usage while maintaining all necessary information, efficient state storage mechanisms optimize memory utilization during the search process, and memory-aware pruning strategies remove unnecessary nodes to conserve memory resources. The implementation also features comprehensive resource tracking capabilities that monitor and optimize resource usage throughout the execution. These implementation features are carefully integrated to create a highly efficient Branch & Bound algorithm that effectively balances search effectiveness with memory efficiency.

5. Key Advantages

- a. Computational Benefits: fast convergence, efficient pruning, limited memory usage, anytime solutions
- b. Solution Quality: Near-optimal results, Guaranteed

- bounds, Solution certificates, Quality guarantees
6. Implementation Trade-offs
 - a. Advantages: Efficient search space exploration, Strong pruning capabilities, Quality guarantees, Memory efficiency
 - b. Limitations: Complex implementation, Variable runtime, Search space dependency, Branch selection impact
 7. Performance Optimization
 - a. Search Strategy: Node selection heuristics, Pruning criteria optimization, Bound computation efficiency, State space management
 - b. Memory Optimization: Node compression, State reuse, Pruning effectiveness, Resource utilization.

2.3 Theoretical complexity analysis

Table 5 presents the theoretical complexity analysis for all implemented algorithms, providing critical insights into their scalability and resource requirements. The complexity is expressed in terms of key problem parameters: n (number of items), m (number of knapsacks), W (maximum weight capacity), B (maximum budget), and p (number of processors for parallel variants).

Table 5. Algorithm complexity comparison

Algorithm	Time Complexity	Space Complexity
BDP	$O(n \times m \times W \times B)$	$O(n \times m \times W \times B)$
Numba	$O(n \times m \times W \times B)$	$O(n \times m \times W \times B)$
Parallel	$O(n \times m \times W \times B)/p$	$O(n \times m \times W \times B)$
P-Rolling	$O(n \times m \times W \times B)$	$O(3 \times m \times W \times B)$
GA	$O(\text{population_size} \times n)$	$O(\text{population_size})$
Greedy	$O(n)$	$O(1)$
Branch & Bound	$O(2^n)^*$	$O(n)$

Note: *worst case

Numba-accelerated implementations maintain the same theoretical complexity $O(n \times m \times W \times B)$ as their traditional counterparts, but achieve significant practical speedup through JIT compilation and hardware optimization. Similarly, the Parallel variant reduces actual runtime by a factor of p through parallel processing, resulting in $O(n \times m \times W \times B)/p$ time complexity, while maintaining the same space requirements.

The parallel rolling variant innovatively reduces space complexity to $O(3 \times m \times W \times B)$ through state space rolling, while maintaining the same time complexity as other DP variants. This represents a significant memory optimization without compromising solution quality.

Genetic algorithm exhibits significantly lower complexity of $O(\text{population_size} \times n)$, making it more scalable for larger problem instances, though without optimality guarantees. The Greedy approach achieves the lowest complexity of $O(n)$ in both time and $O(1)$ in space, offering extremely fast execution at the cost of solution quality.

Branch & Bound, while having worst-case time complexity of $O(2^n)$, often performs significantly better in practice due to effective pruning strategies. Its space complexity remains linear at $O(n)$, making it memory-efficient for larger problems.

The theoretical complexity analysis provides insights into algorithm scalability and resource requirements. Notably, while Branch & Bound shows exponential worst-case complexity, its practical performance can be significantly better due to effective pruning strategies. The parallel variants maintain the same theoretical complexity as BDP but achieve

practical speedup through concurrent execution.

2.4 Evaluation framework

2.4.1 Experimental setup

All experiments were conducted on Intel Core Ultra 7 22 threads with 32 GB memory running Python 3.8. For reproducibility, random seeds were set to 42 where applicable. The implementation utilized NumPy 1.19 for numerical computations and Pandas 1.2 for data management. Tests were executed with a time limit of 3600 seconds per algorithm.

Test Dataset Characteristics as follows: 88 products (45 beverages, 43 snacks); 7 knapsacks with varying constraints; Value range: 1.0-10.0 units; Weight range: 0.11-1.0 kg; Price range: 6,000-60,000 IDR.

The knapsack constraints were designed to reflect real-world retail bundling scenarios, with capacities ranging from 1.5kg to 4.0kg and budgets from 150,000 to 300,000 IDR.

2.4.2 Performance metrics

The performance evaluation framework incorporates multiple metric categories to ensure comprehensive assessment of each algorithm. For solution quality assessment, we employ Total Value Achievement (TVA) to measure absolute solution value, alongside Relative Value Achievement (RVA) which contextualizes solution value against theoretical optimum. Resource Utilization Efficiency (RUE – denoted as price utilization ratio) evaluates how effectively each algorithm utilizes available constraints, while Constraint Violation Rate (CVR) monitors the frequency of constraint breaches during solution construction.

In terms of computational efficiency, we track Execution Time (ET) measured in CPU seconds, alongside detailed memory profiling through Peak Memory Usage (PMU) and Average Memory Consumption (AMC) metrics measured in megabytes. The framework also considers Scaling Efficiency to evaluate performance changes with increasing problem size, and employs a Resource Efficiency Ratio (RER) to provide a balanced measure of solution quality versus computational cost. These metrics collectively provide a multi-dimensional view of algorithm performance, enabling thorough comparative analysis.

$$RVA = \frac{TVA}{Theoretical_Maximum} \quad (6)$$

$$Price\ utilization\ ratio = \frac{used\ price}{available\ budget} \quad (7)$$

$$RER = \frac{TVA}{(ET \times PMU)} \quad (8)$$

Additionally, we incorporate algorithm-specific metrics to capture unique characteristics of each implementation. These include convergence rate analysis to measure solution improvement over time, solution stability assessment to evaluate result consistency across multiple runs, and population diversity metrics specifically for the genetic algorithm implementation to monitor solution space exploration.

2.4.3 Statistical analysis framework

To facilitate objective comparison, we implement a Multi-criteria Decision Analysis using the TOPSIS method for

algorithm ranking. The criteria weights are carefully distributed to reflect practical importance: solution quality (0.5), runtime efficiency (0.3), memory usage (0.1), and resource utilization (0.1). This weighting scheme prioritizes solution quality while maintaining balanced consideration of computational efficiency and resource utilization aspects. Moreover, we are using 10% weight perturbation in matter of stability verification.

3. RESULTS AND ANALYSIS

3.1 Solution quality assessment

Our comprehensive analysis demonstrates that five algorithms consistently achieve optimal solutions, while three algorithms provide high-quality approximate solutions. Table 6 presents the detailed performance comparison across all implementations.

Table 6. Solution quality comparison

Algorithm	TVA	Price Util (%)	Weight Util (%)	RER	Memory Eff.(%)	Overall Score*	Constraint Violations
Parallel	247.50	99.66	94.46	204.58	46.86	0.890	0
P-Rolling	247.50	99.66	81.08	221.73	82.59	0.928	0
Numba	247.50	99.66	81.08	609.91	29.05	0.683	0
BDP	247.50	99.66	81.08	3.13	64.35	0.262	0
GA	230	92.62	83.90	142.73	172.39	0.883	0
Greedy	232.50	93.62	58.05	232500	157.67	0.880	0
Branch & Bound	223.50	90.00	71.64	5359.71	189.26	0.845	0

* Overall score based on TOPSIS analysis incorporating all metrics

Table 7. Computational efficiency

Algorithm	Runtime (s)	Peak Memory (MB)
Parallel	1.2098	528.11
P-Rolling	1.1162	299.68
Numba	0.4058	852.08
BDP	79.1570	384.63
GA	1.6823	133.46
Greedy	0.0000	147.46
Branch & Bound	0.0417	118.09

The Resource Efficiency Ratio (RER) reveals interesting performance patterns. The greedy algorithm achieved an exceptionally high RER of 232500, primarily due to its negligible runtime. Branch & Bound also showed strong efficiency with an RER of 5359.71. Numba demonstrated good balance with an RER of 609.91, while BDP showed lower ratios of 3.13 respectively.

Memory efficiency varied across implementations, with Branch & Bound showing the highest efficiency at 189.26%, followed by GA at 172.39%. The Numba implementation demonstrated the lowest memory efficiency at 29.05%.

The Overall Score, based on TOPSIS analysis incorporating all metrics, ranks P-Rolling highest at 0.928, followed by parallel processing at 0.890. BDP ranked lowest with a score of 0.262. Notably, no algorithm recorded any constraint violations, indicating robust implementation of feasibility checks across all approaches. This comprehensive comparison demonstrates the trade-offs between solution quality, resource utilization, and computational efficiency across different algorithmic approaches to the MKP.

Table 7 presents a comprehensive analysis of computational performance across eight algorithm implementations, focusing on two critical metrics: Runtime (in seconds) and Peak

Memory Usage (in MB). The results reveal distinct patterns in computational efficiency and resource utilization.

Runtime Performance: (1) The greedy algorithm demonstrates exceptional speed with a negligible runtime of 0.0000 seconds; (2) Branch & Bound shows remarkable efficiency with just 0.0417 seconds; (3) Numba-accelerated implementation achieves impressive performance at 0.4058 seconds; (4) Parallel and P-Rolling variants maintain good efficiency at 1.2098 and 1.1162 seconds respectively; (5) The genetic algorithm (GA) requires slightly more time at 1.6823 seconds; (6) BDP approaches show significantly longer execution times requires 79.1570.

Peak Memory Usage: (1) Branch & Bound demonstrates the most efficient memory utilization at 118.09 MB; (2) GA, and Greedy show moderate memory consumption as GA: 133.46 MB, and Greedy: 147.46 MB; (3) P-Rolling achieves balanced memory usage at 299.68 MB; (4) BDP requires increased memory at 384.63 MB; (5) Higher memory requirements are seen in Parallel: 528.11 MB and Numba: 852.08 MB.

Thus, Memory consumption patterns revealed three distinct tiers:

1. Low Memory Tier (<150MB)
 - a. Branch & Bound: 118.09MB
 - b. GA: 133.46MB
 - c. Greedy: 147.46MB
2. Medium Memory Tier (150-400MB)
 - a. P-Rolling: 299.68MB
 - b. BDP: 384.63MB
3. High Memory Tier (>400MB)
 - a. Parallel: 528.11MB
 - b. Numba: 852.08MB

The results highlight clear trade-offs between execution speed and memory consumption. While some algorithms

achieve faster runtimes, they may require more memory resources, exemplifying the classic space-time trade-off in algorithm design. This comparison provides crucial insights for implementation decisions based on available computational resources and performance requirements.

3.2 Computational performance

Table 8 provides a detailed comparison of runtime performance across seven algorithms, using BDP as the baseline for speedup calculations.

Table 8. Runtime performance summary

Algorithm	Runtime(s)	Speedup vs Base
BDP	79.1570	1.00x (baseline)
Numba	0.4058	240.14x
Parallel	1.2098	80.47x
P-Rolling	1.1162	87.21x
GA	1.6823	57.87x
Greedy	0.0000*	N/A
Branch & Bound	0.0417	2334.53x

*Below measurable threshold

The results demonstrate significant variations in computational efficiency:

Baseline Performance: BDP establishes the baseline with a runtime of 79.1570 seconds, this serves as the reference point (1.00x speedup) for comparing other implementations.

Relative Performance Improvements: (1) Numba: Runtime: 0.4058 seconds, Achieves remarkable 240.14x speedup, Demonstrates the effectiveness of JIT compilation optimization; (2) Parallel: Runtime: 1.2098 seconds, Achieves 80.47x speedup, Shows significant benefits of parallel processing; (3) P-Rolling: Runtime: 1.1162 seconds, Achieves 87.21x speedup, Indicates efficiency of combined parallel and memory optimization; (4) GA: Runtime: 1.6823 seconds, Achieves 57.87x speedup, Shows competitive performance for a metaheuristic approach; (5) Greedy: Runtime: 0.0000* seconds, Speedup: Not applicable due to negligible runtime, Demonstrates exceptional computational efficiency; (6) Branch & Bound: Runtime: 0.0417 seconds, Achieves

impressive 2334.53x speedup, Shows remarkable efficiency for an exact method.

This comparison reveals a clear hierarchy in computational efficiency, with modern optimization techniques (Branch & Bound, Numba) significantly outperforming base approaches. The results highlight the substantial impact of algorithm choice on runtime performance in solving the MKP.

Then, we utilized TOPSIS rankings as seen in Table 9.

Table 9. TOPSIS rankings with detailed scores

Algorithm	Topsis Score	Performance Breakdown			
		Solution Quality	Runtime Efficiency	Memory Usage	Resource Utilization
Parallel	0.890	1.000	0.987	0.620	0.971
P-Rolling	0.928	1.000	0.989	0.716	0.903
Numba	0.683	1.000	0.996	0.000	0.903
BDP	0.262	1.000	0.187	0.549	0.903
GA	0.883	0.930	0.983	0.842	0.883
Greedy	0.880	0.940	1.000	0.825	0.758
Branch & Bound	0.845	0.903	0.999	0.858	0.808

Note: All scores normalized to [0,1] range where 1.0 represents best performance

As seen in Table 9, The TOPSIS analysis, incorporating multiple performance criteria weighted according to practical importance, clearly identifies P-Rolling as the superior approach (score: 0.928). This ranking considers not only solution quality but also runtime efficiency, memory usage, and resource utilization patterns. Notably, while some algorithms achieved perfect scores in individual categories (e.g., Greedy for runtime efficiency), the parallel variants demonstrated the best overall balance of performance metrics.

Particularly noteworthy is the clustering of scores, with parallel variants (Parallel and P-Rolling) forming a high-performance tier (scores > 0.880), followed by a middle tier of optimized implementations (scores 0.800-0.880), and a lower tier of basic implementations (scores < 0.800). This tiering suggests clear implementation strategy recommendations for different use cases. To justify the TOPSIS, analysis, Table 10 represent 10% perturbation as follows:

Table 10. TOPSIS stability verification

Algorithm	Ori Score	Scenario1	Scenario2	Scenario3	Scenario4	Scenario 5	Mean	Std	CV
Parallel	0.890	0.886	0.898	0.892	0.885	0.897	0.891	0.005	0.006
P-Rolling	0.928	0.925	0.933	0.929	0.925	0.931	0.929	0.003	0.003
Numba	0.683	0.673	0.704	0.689	0.670	0.701	0.687	0.013	0.019
BDP	0.262	0.270	0.241	0.254	0.273	0.243	0.257	0.012	0.048
GA	0.883	0.881	0.895	0.891	0.880	0.896	0.888	0.006	0.007
Greedy	0.880	0.880	0.891	0.885	0.878	0.887	0.884	0.005	0.005
Branch & Bound	0.845	0.843	0.861	0.885	0.842	0.861	0.851	0.008	0.009

* Average CV: 0.014 and Maximum score variation: ±0.013

From Table 10, we can observe that the P-Rolling and Parallel obtain high score than other algorithm. Moreover, the empirical results largely align with theoretical complexity predictions:

1. Memory Usage Patterns: P-Rolling achieved constant memory scaling ($O(3 \times m \times W \times B)$) as predicted; Parallel variants showed linear scaling with problem size; GA maintained population-size bounded memory usage
2. Runtime Behavior: Greedy's linear complexity reflected in fastest execution; Parallel variants achieved near-linear speedup with processor count; Branch & Bound

performed better than worst-case bounds suggest The correlation between theoretical and observed performance validates the implementation efficiency of each algorithm.

3.3 Scalability analysis

To assess the practical applicability of our algorithms across different operational scales, we conducted a systematic scalability analysis using subsets of our dataset. Starting from the base dataset of 88 items, we created controlled test scenarios with 17, 41, 65, and 88 items while maintaining the

original data characteristics and knapsack constraints. This approach allows us to evaluate performance scaling patterns and resource utilization trends across different problem sizes, providing insights into each algorithm's behavior as

computational demands increase. Our analysis focuses on three critical metrics: execution time, memory consumption, and solution quality maintenance.

Table 11. Execution time scalability analysis

Size	BDP	Numba	Parallel	P-Rolling	GA	Greedy	Branch & Bound
17	48.97	0.19	0.88	0.76	0.04	0000	0.01
41	23.57	0.25	0.89	0.81	0.44	0.00	0.01
65	29.35	0.27	0.79	0.93	0.72	0.00	0.01
88	79.15	0.40	1.21	1.12	1.68	0.00	0.04

Table 12. Memory consumption scalability analysis

Size	BDP	Numba	Parallel	P-Rolling	GA	Greedy	Branch & Bound
17	365.97	286.30	234.96	266.67	173.49	158.44	167.07
41	240.08	364.96	364.92	283.37	183.51	172.08	167.07
65	290.67	439.83	428.48	292.91	171.93	164.37	160.52
88	384.63	852.08	528.11	299.68	133.46	147.46	118.09

Table 13. Solution quality maintenance analysis

Size	BDP	Numba	Parallel	P-Rolling	GA	Greedy	Branch & Bound
17	247.5	247.5	242.5	247.5	99	99	195
41	247.5	247.5	247.5	247.5	198.5	193.5	195.0
65	247	247.0	247.5	247.5	227.5	222	226.5
88	247.5	247.5	247.5	247.5	230	232.5	223.5

The scalability analysis reveals distinct performance patterns across different problem sizes, providing crucial insights into algorithm behavior under varying computational demands. Our systematic evaluation examined four dataset scales (17, 41, 65, and 88 items) while maintaining consistent knapsack constraints and data characteristics.

Execution time analysis (Table 11) demonstrates notable scaling patterns: 1) Parallel variants (Parallel and P-Rolling) maintain relatively stable performance across different problem sizes, with execution times ranging from 0.76-1.21 seconds, demonstrating efficient resource utilization even as problem complexity increases; 2) The BDP implementation shows significant performance degradation with increased problem size, with execution time increasing from 48.97 to 79.15 seconds; 3) Numba-accelerated implementation maintains impressive efficiency, showing only modest increases in execution time (0.19 to 0.40 seconds) despite problem size quadrupling; 4) Branch & Bound and greedy algorithms demonstrate remarkable stability, with execution times remaining consistently low across all problem sizes.

Memory consumption patterns (Table 12) reveal important resource utilization characteristics: 1) Parallel rolling demonstrates superior memory efficiency, maintaining relatively stable memory usage (266.67-299.68 MB) across problem sizes; 2) BDP and Numba show more pronounced memory scaling, with Numba's consumption increasing significantly from 286.30 MB to 852.08 MB for the largest problem size; 3) GA and Branch & Bound maintain conservative memory profiles, actually showing slight decreases in memory usage for larger problem sizes, suggesting effective memory management strategies.

Solution quality maintenance (Table 13) provides critical insights into algorithm reliability: 1) Exact methods (BDP, Numba, Parallel, P-Rolling) consistently achieve optimal solutions (247.5) across all problem sizes; 2) Metaheuristic approaches (GA) show improving solution quality with

increased problem size, from 99.0 to 230.0; 3) Branch & Bound maintains relatively stable solution quality (195.0-223.5) despite increasing problem complexity; 4) Greedy algorithm demonstrates improving solution quality with larger problem sizes, reaching 232.5 for the 88-item case.

This comprehensive scalability analysis validates the practical applicability of our implementations across different operational scales, with parallel variants demonstrating particularly robust performance characteristics. The analysis confirms that algorithm selection should consider not only absolute performance metrics but also scaling behavior relative to expected problem sizes and available computational resources.

These findings extend our understanding of algorithmic behavior beyond theoretical complexity analysis, providing practical insights for implementation decisions across different operational scales. The observed patterns support our implementation recommendations, particularly the adoption of parallel variants for large-scale operations and Branch & Bound or Greedy approaches for resource-constrained environments.

4. DISCUSSION

4.1 Algorithm performance insights

The comprehensive evaluation of eight MKP algorithms reveals distinct performance tiers and implementation trade-offs. The parallel rolling (P-R) achieved optimal solution quality (247.50) with resource utilization (99.66% price, 81.08% weight). Followed by the parallel processing (P-P) variants consistently demonstrated superior performance across multiple metrics, achieving optimal solution quality (247.50) while maintaining high resource utilization (99.66% price, 94.46% weight). Even though, P-R obtain lower in weight

utilization, it does outperform in matter of TOPSIS analysis followed by P-P. Thus, P-R and P-P is interchangeable according to the user needs.

4.1.1 Performance-resource trade-offs

Our analysis reveals distinct performance tiers among the implemented algorithms. The High-Performance Tier, dominated by parallel rolling and parallel processing variants, demonstrates exceptional capabilities with optimal solution quality achieving 99.66% price utilization and 94.46% weight utilization. These implementations deliver significant performance improvements, showing 80.47x speedup compared to BDP. While they incur higher memory overhead, the performance gains justify this trade-off, making them: 1) ideal for large-scale retail operations handling multiple product lines; 2) Supports real-time decision making in dynamic pricing scenarios; 3) Enables efficient resource allocation in complex bundling strategies. However, these implementations require specific infrastructure considerations, including multi-core processor capabilities, substantial memory allocation, and careful thread management overhead handling.

The Balanced Performance Tier, represented by Numba and Branch & Bound implementations, offers a compelling compromise between performance and resource requirements. These algorithms consistently achieve near-optimal solutions exceeding 90% of optimal value, while delivering excellent runtime efficiency with over 240x speedup. Their moderate memory requirements, ranging from 118.09 to 852.08 MB, coupled with predictable performance characteristics and stable resource utilization, make them 1) Suitable for medium-scale retail operations; 2) Enables quick response to market changes; 3) Supports efficient inventory management.

In the resource-efficient tier, the greedy and genetic algorithm implementations provide practical solutions for resource-constrained environments. These algorithms maintain good solution quality exceeding 93% of optimal value while operating with minimal memory footprint below 150MB. Their linear scaling characteristics make them 1) Ideal for small retailers with limited IT infrastructure; 2) Supports rapid decision-making in straightforward bundling scenarios; 3) Cost-effective implementation.

4.1.2 Implementation considerations

The choice of algorithm significantly impacts both solution quality and resource requirements. Memory usage varies from 118.09MB (Branch & Bound) to 852.08MB (Numba), while runtime ranges from negligible (Greedy) to 79.1570 seconds (BDP).

4.2 Business context analysis

4.2.1 Enterprise retail applications

For large retail operations, our findings support the adoption of parallel processing solutions based on: (1) Consistent achievement of optimal solutions; (2) High resource utilization (>94% for both constraints); (3) Linear scaling with processor count; (4) Robust performance across problem variations.

Alternatively, enterprise retail applications can adopt parallel rolling based on: memory-constrained environments, similar performance benefits, reduced memory footprint (299.68 MB).

4.2.2 SME implementation considerations

Small-medium enterprises benefit from lighter-weight implementations: (1) Branch & Bound provides 90% utilization with minimal resources; (2) Greedy algorithm offers immediate results for time-critical decisions; (3) Memory efficiency crucial for limited infrastructure; (4) Acceptable solution quality for smaller scale operations.

Alternatively, SME could use Greedy Algorithm where Time-critical operations, Limited computational resources, Straightforward implementation.

4.2.3 Implementation strategy framework

Our Implementation Strategy Framework provides comprehensive guidelines tailored to different operational scales, based on empirical evidence from our algorithmic analysis. For enterprise-scale implementations, we recommend deploying parallel variants, particularly parallel rolling, which demonstrated superior performance (TOPSIS score: 0.928) and optimal resource utilization (99.66% price, 81.08% weight utilization). The implementation requires specific technical infrastructure including multi-core processors, minimum 16GB RAM, and appropriate storage systems. The deployment process follows a structured three-phase approach: infrastructure assessment and setup, algorithm deployment with parallel processing configuration, and performance optimization through thread allocation tuning and cache optimization.

For SME implementations, we propose a resource-conscious approach centered on Branch & Bound algorithm, which achieved exceptional efficiency (0.0417 seconds runtime) while maintaining minimal resource requirements (118.09 MB memory footprint). This implementation pathway requires standard computing infrastructure and follows a simplified three-phase deployment: basic setup verification, streamlined algorithm implementation, and targeted performance tuning. Both frameworks are supported by empirical evidence from our experimental results, ensuring practical applicability while maintaining solution quality across different operational scales. The implementation guidelines incorporate specific technical requirements, algorithmic selection criteria based on TOPSIS analysis, and detailed deployment phases, providing a comprehensive roadmap for successful implementation across varying business contexts.

4.3 Theoretical, algorithm adaptability and practical implications

Our study reveals several significant insights into algorithm design and implementation. The analysis demonstrates that memory-performance trade-offs play a crucial role in determining practical utility, while parallel processing approaches effectively address computational bottlenecks in large-scale implementations. Notably, simple heuristic approaches have shown remarkable capability in providing competitive solutions for practical applications, challenging the notion that complex algorithms are always necessary for effective problem-solving.

Our analysis reveals critical insights into algorithm adaptability across dynamic operational environments. The parallel variants demonstrate exceptional resilience under parameter variations, maintaining solution quality within 98.5% of optimal values during price fluctuations ($\pm 20\%$) and achieving >95% resource utilization efficiency under weight

variations ($\pm 15\%$). These implementations maintain consistent runtime performance (1.1162-1.2098 seconds) and stable memory utilization ($<5\%$ variation), validating their suitability for enterprise-scale deployments where parameter stability cannot be guaranteed. Similarly, the Branch & Bound algorithm exhibits robust adaptability characteristics particularly valuable for resource-constrained environments, maintaining $>90\%$ solution quality under price variations while preserving efficient execution time (0.0417 seconds $\pm 8\%$) and stable memory consumption (118.09 MB $\pm 5\%$), with Greedy and GA implementations showing predictable performance patterns under dynamic conditions.

This empirical evidence substantiates our implementation recommendations across varying operational scenarios, with parallel variants demonstrating superior adaptability for enterprise deployments and Branch & Bound emerging as a reliable choice for SME applications. The observed performance stability under dynamic parameter changes validates both the theoretical robustness of our algorithmic approaches and their practical applicability in real-world retail environments where price and weight variations are common operational challenges. These findings significantly enhance our understanding of algorithm behavior under dynamic conditions, providing crucial insights for implementation decisions while maintaining optimal resource utilization and solution quality across different operational scales.

The implementation patterns identified through this research highlight the importance of sophisticated resource allocation strategies. Successful implementations require careful attention to dynamic memory management for large problem instances, optimization of processor allocation, and implementation of cache-aware data structures. These technical considerations are complemented by robust solution quality management approaches, including the establishment of early termination criteria based on solution quality thresholds, careful balancing of resource utilization, and comprehensive monitoring of constraint satisfaction throughout the solution process.

This research contributes significantly to the theoretical understanding and practical application of MKP algorithms. By quantifying performance-resource trade-offs across different implementation scenarios, validating the benefits of parallel processing approaches in MKP solutions, and establishing clear implementation guidelines for varying operational scales, our findings provide valuable insights for both academic research and practical implementations. These implications extend beyond the immediate context of retail applications, offering broader perspectives on algorithm selection and implementation strategies for complex optimization problems.

4.4 Limitations and considerations

This study acknowledges several limitations that provide context for result interpretation and future research directions. The dataset characteristics present certain constraints, as our analysis is based on a specific set of 88 items distributed across 7 knapsacks, with fixed constraint ranges tailored to retail product bundling scenarios. While this dataset effectively represents typical retail bundling problems, it may not capture all possible real-world scenarios. The categorical distribution (45 beverages, 43 snacks) and value ranges (1.0-10.0 units for value coefficients, 0.11-1.0 kilograms for weights, 6,000-60,000 IDR for prices) provide comprehensive coverage for

traditional retail scenarios but may require validation for extreme price points or highly specialized product categories, such as: seasonal variations in product availability, regional price fluctuations, dynamic inventory patterns, and extreme price point products.

Performance evaluation boundaries are defined by our experimental setup, including memory measurements conducted on specific hardware configurations, the use of single-node parallel processing architectures, and implementation-specific optimizations. These boundaries, while providing consistent comparative analysis, may influence the direct applicability of results to different hardware environments. The generalization of our findings is most relevant to retail product bundling scenarios, particularly those involving dual-constraint knapsack problems of similar scale and complexity. While our scalability analysis demonstrates consistent performance across different problem sizes (17-88 items) and our statistical validation shows solution stability (± 0.013 TOPSIS score variation under 10% weight perturbation), the applicability to significantly larger retail operations (>1000 items) or different market segments may require additional validation.

4.5 Implementation performance and complexity analysis

Our comprehensive analysis of implementation performance reveals distinct patterns across algorithmic approaches. The theoretical complexity analysis demonstrates clear efficiency tiers: parallel variants achieve $O(n \times m \times W \times B) / p$ time complexity with processor count p , while maintaining $O(3 \times m \times W \times B)$ space complexity through rolling optimization; metaheuristic approaches show $O(\text{population_size} \times n)$ time and $O(\text{population_size})$ space complexity; and resource-efficient methods like Greedy maintain $O(n)$ time with $O(1)$ space requirements. These theoretical bounds are validated by empirical results showing parallel variants achieving 80.47x-87.21x speedup over BDP, Numba demonstrating 240.14x improvement, and Branch & Bound exhibiting exceptional efficiency with 2334.53x speedup, while maintaining solution quality above 90% of optimal value across implementations.

Resource utilization patterns further support these findings, with price utilization ranging from 90.00% to 99.66% and weight utilization varying from 58.05% to 94.46%. Memory consumption remains well-controlled, particularly in resource-efficient implementations like Branch & Bound (118.09 MB) and GA (<150 MB), while parallel variants demonstrate predictable scaling with problem size. These empirical results, combined with consistent solution quality maintenance, validate our implementation strategies and provide robust criteria for deployment decisions across different operational scales, eliminating the need for extensive cross-platform testing while maintaining rigorous performance validation.

4.6 Future research opportunities

Looking forward, several promising research opportunities emerge from this work. In terms of algorithm enhancement, there is significant potential for developing hybrid approaches that combine the strengths of multiple algorithms, implementing dynamic parameter adaptation mechanisms for improved performance, and exploring advanced resource optimization strategies. The implementation domain could be expanded through investigation of distributed processing

architectures, exploration of cloud-based deployment options, and development of real-time optimization capabilities. These future directions could significantly advance the practical application of MKP solutions in retail and beyond.

5. CONCLUSION

This comprehensive study of MKP algorithms reveals significant findings regarding algorithm effectiveness and implementation considerations. Our analysis demonstrates that parallel variant implementations achieve optimal solutions with a value of 247.50 and exceptional resource utilization of 99.66%, setting a new benchmark for performance. Notably, Branch & Bound and Greedy algorithms emerge as efficient alternatives for resource-constrained environments, offering practical solutions with reduced computational requirements.

The performance-resource trade-offs observed across implementations provide valuable insights for practical deployment. Memory requirements vary substantially, ranging from 118.09MB to 852.08MB across different implementations, while runtime optimizations achieve impressive improvements of up to 240x compared to baseline approaches. Significantly, top-performing algorithms consistently maintain resource utilization above 90%, demonstrating robust efficiency in constraint satisfaction.

Our TOPSIS analysis reveals a clear tiering of implementations based on performance characteristics. High-performance solutions, led by P-Rolling (0.928) and parallel processing (0.890), deliver optimal results with superior resource utilization. Resource-efficient options, comprising GA/Greedy (0.883/0.880) and Branch & Bound (0.845), provide practical solutions for environments with limited computational resources.

This study makes several significant contributions: (1) Methodological Advances presents Comprehensive evaluation framework for dual-constrained MKP, Statistical validation approach for algorithm comparison, Implementation guidelines for different operational scales; (2) Practical Applications for Enterprise Solutions are Evidence-based algorithm selection criteria for retail applications, Resource requirement quantification for implementation planning, Performance expectations for different operational scales; (3) Practical Applications for SME solutions are Resource-efficient algorithm recommendations, Scalable solution pathways, Cost-effective deployment strategies.

Several promising areas for future investigation emerge: (1) Algorithm Enhancement such as Hybrid approaches combining metaheuristic and exact methods, Advanced parallelization strategies for distributed systems, Adaptive parameter tuning for metaheuristic algorithms; (2) Implementation Extensions such as Cloud-based deployment strategies, Real-time optimization capabilities, Integration with inventory management systems; (3) Scalability Analysis such as Investigation of larger problem instances, Multi-node parallel processing evaluation, Dynamic constraint handling mechanisms.

The findings support specific recommendations for implementation: (1) for Enterprise Scale, Implement parallel processing solutions for optimal performance, Utilize multi-core architectures for computational efficiency, Focus on high resource utilization capabilities; (2) for SME Applications, Deploy Branch & Bound for balanced performance, Consider

Greedy algorithms for time-critical decisions, Optimize for minimal resource requirements; (3) for General Guidelines, Match algorithm selection to available computational resources, Consider scaling requirements in implementation planning, Balance solution quality with operational constraints.

This research demonstrates the effectiveness of parallel processing approaches for solving dual-constrained MKP in retail contexts. While parallel variants achieve optimal solutions with high resource utilization, simpler algorithms provide viable alternatives for resource-constrained environments. The findings provide a foundation for implementing MKP solutions across different operational scales while considering practical constraints and requirements.

The established evaluation framework and implementation guidelines contribute to both theoretical understanding and practical application of MKP algorithms in retail product bundling. Future research can build on these findings to develop more sophisticated solutions for larger-scale problems and specialized retail applications.

ACKNOWLEDGMENT

This work was supported by the Kemdikbud Research Grant on Penelitian Fundamental-Reguler with grant number 108/E5/PG.02.00.PL/2024 and 027/LL6/PB/AL.04/2024; 061/A38-04/UDN-09/VI/2024.

REFERENCES

- [1] Dell'Amico, M., Delorme, M., Iori, M., Martello, S. (2019). Mathematical models and decomposition methods for the multiple knapsack problem. *European Journal of Operational Research*, 274(3): 886-899. <https://doi.org/10.1016/j.ejor.2018.10.043>
- [2] Mkaouar, A., Htiouech, S., Chabchoub, H. (2023). Modified artificial bee colony algorithm for multiple-choice multidimensional knapsack problem. *IEEE Access*, 11: 45255-45269. <https://doi.org/10.1109/ACCESS.2023.3264966>
- [3] Lalonde, O., Côté, J.F., Gendron, B. (2022). A branch-and-price algorithm for the multiple knapsack problem. *INFORMS Journal on Computing*, 34(6): 3134-3150. <https://doi.org/10.1287/ijoc.2022.1223>
- [4] Caserta, M., Voß, S. (2019). The robust multiple-choice multidimensional knapsack problem. *Omega*, 86: 16-27. <https://doi.org/10.1016/j.omega.2018.06.014>
- [5] Bansal, S., Patvardhan, C. (2018). An improved generalized quantum-inspired evolutionary algorithm for multiple knapsack problem. *International Journal of Applied Evolutionary Computation (IJAEC)*, 9(1): 17-51. <https://doi.org/10.4018/978-1-7998-8593-1.ch002>
- [6] Zhao, Q., Tan, K., Du, J., Van Woensel, T. (2023). Joint case pack size and unpacking location optimization in a retail supply chain including product returns. *Computers & Industrial Engineering*, 182: 109415. <https://doi.org/10.1016/j.cie.2023.109415>
- [7] Gecili, H., Parikh, P.J. (2022). Joint shelf design and shelf space allocation problem for retailers. *Omega*, 111: 102634. <https://doi.org/10.1016/j.omega.2022.102634>
- [8] Deza, A., Huang, K., Liang, H., Wang, X.J. (2020). On inventory allocation for periodic review assemble-to-

- order systems. *Discrete Applied Mathematics*, 275: 29-41. <https://doi.org/10.1016/j.dam.2019.04.004>
- [9] Ortega, F.A., Mesa, J.A., Piedra-De-La-Cuadra, R., Pozo, M.A. (2019). A matheuristic for optimizing skip–stop operation strategies in rail transit lines. *International Journal of Transport Development and Integration*, 3(4): 306-316. <https://doi.org/10.2495/TDI-V3-N4-306-316>
- [10] Ikhelef, A., Saidi, M.Y., Li, S., Chen, K. (2022). A knapsack-based optimization algorithm for VNF placement and chaining problem. In 2022 IEEE 47th Conference on Local Computer Networks (LCN), Edmonton, AB, Canada, pp. 430-437. <https://doi.org/10.1109/LCN53696.2022.9843566>
- [11] Simon, J., Apte, A., Regnier, E. (2017). An application of the multiple knapsack problem: The self-sufficient marine. *European Journal of Operational Research*, 256(3): 868-876. <https://doi.org/10.1016/j.ejor.2016.06.049>
- [12] Gu, Z., Lu, H., Zhu, D., Lu, Y. (2018). Joint power allocation and caching optimization in fiber-wireless access networks. In 2018 IEEE Global Communications Conference (GLOBECOM), Abu Dhabi, United Arab Emirates, pp. 1-7. <https://doi.org/10.1109/GLOCOM.2018.8647800>
- [13] Jiang, Q., Zhang, Y., Yan, J. (2020). Neural combinatorial optimization for energy-efficient offloading in mobile edge computing. *IEEE Access*, 8: 35077-35089. <https://doi.org/10.1109/ACCESS.2020.2974484>
- [14] Schäfer, G., Zweers, B.G. (2021). Maximum coverage with cluster constraints: An LP-based approximation technique. In *Approximation and Online Algorithms: 18th International Workshop, WAOA 2020, Virtual Event*, pp. 63-80. https://doi.org/10.1007/978-3-030-80879-2_5
- [15] Babukarthik, R.G., Dhasarathan, C., Kumar, M., Shankar, A., Thakur, S., Cheng, X. (2021). A novel approach for multi-constraints knapsack problem using cluster particle swarm optimization. *Computers & Electrical Engineering*, 96: 107399. <https://doi.org/10.1016/j.compeleceng.2021.107399>
- [16] Fidanova, S., Atanassov, K.T. (2021). ACO with intuitionistic fuzzy pheromone updating applied on multiple-constraint knapsack problem. *Mathematics*, 9(13): 1456. <https://doi.org/10.3390/math9131456>
- [17] Wu, Q., He, M., Hao, J.K., Lu, Y. (2024). An effective hybrid evolutionary algorithm for the clustered orienteering problem. *European Journal of Operational Research*, 313(2): 418-434. <https://doi.org/10.1016/j.ejor.2023.08.006>
- [18] Lai, X., Hao, J.K., Fu, Z.H., Yue, D. (2020). Diversity-preserving quantum particle swarm optimization for the multidimensional knapsack problem. *Expert Systems with Applications*, 149: 113310. <https://doi.org/10.1016/j.eswa.2020.113310>
- [19] Chen, Y., Hao, J.K., Glover, F. (2016). An evolutionary path relinking approach for the quadratic multiple knapsack problem. *Knowledge-Based Systems*, 92: 23-34. <https://doi.org/10.1016/j.knosys.2015.10.004>
- [20] Areias, M., Rocha, R. (2017). On scaling dynamic programming problems with a multithreaded tabling Prolog system. *Journal of Systems and Software*, 125: 417-426. <https://doi.org/10.1016/j.jss.2016.06.060>
- [21] Salhab, N., Rahim, R., Langar, R. (2018). Throughput-aware RRHs clustering in cloud radio access networks. In 2018 Global Information Infrastructure and Networking Symposium (GIIS), Thessaloniki, Greece, pp. 1-5. <https://doi.org/10.1109/GIIS.2018.8635647>
- [22] Boukhari, S., Dahmani, I., Hifi, M. (2022). Computational power of a hybrid algorithm for solving the multiple knapsack problem with setup. In *Intelligent Computing: Proceedings of the 2021 Computing Conference*, pp. 154-168. https://doi.org/10.1007/978-3-030-80119-9_7
- [23] Fidanova, S. (2020). Hybrid ant colony optimization algorithm for multiple knapsack problem. In 2020 5th IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE), Jaipur, India, pp. 1-5. <https://doi.org/10.1109/ICRAIE51050.2020.9358351>
- [24] Zulfa, M.I., Hartanto, R., Permanasari, A.E., Ali, W. (2022). Improving cached data offloading optimization based on enhanced hybrid ant colony genetic algorithm. *IEEE Access*, 10: 84558-84568. <https://doi.org/10.1109/ACCESS.2022.3197205>
- [25] Li, X., Liu, S., Wang, J., Chen, X., Ong, Y.S., Tang, K. (2024). Chance-constrained multiple-choice knapsack problem: Model, algorithms, and applications. *IEEE Transactions on Cybernetics*, 54(12): 7969-7980. <https://doi.org/10.1109/TCYB.2024.3402395>
- [26] Jovanovic, R., Voß, S. (2024). Matheuristic fixed set search applied to the multidimensional knapsack problem and the knapsack problem with forfeit sets. *OR Spectrum*, 46: 13229-1365. <https://doi.org/10.1007/s00291-024-00746-2>
- [27] Paul, J., Agatz, N., Spliet, R., De Koster, R. (2019). Shared capacity routing problem—An omni-channel retail study. *European Journal of Operational Research*, 273(2): 731-739. <https://doi.org/10.1016/j.ejor.2018.08.027>
- [28] Wang, J., Liu, T., Liu, K., Kim, B., Xie, J., Han, Z. (2018). Computation offloading over fog and cloud using multi-dimensional multiple knapsack problem. In 2018 IEEE Global Communications Conference (GLOBECOM), Abu Dhabi, United Arab Emirates, pp. 1-7. <https://doi.org/10.1109/GLOCOM.2018.8647854>
- [29] Ayenew, T.M., Xenakis, D., Passas, N., Merakos, L. (2021). Cooperative content caching in MEC-enabled heterogeneous cellular networks. *IEEE Access*, 9: 98883-98903. <https://doi.org/10.1109/ACCESS.2021.3095356>
- [30] Moryadee, C., Aunyawong, W., Shaharudin, M.R. (2019). Congestion and pollution, vehicle routing problem of a logistics provider in Thailand. *The Open Transportation Journal*, 13(1): 203-212. <https://doi.org/10.2174/1874447801913010203>
- [31] Cerulli, R., D'Ambrosio, C., Raiconi, A. (2024). A biased random-key genetic algorithm for the knapsack problem with forfeit sets. *Soft Computing*, 28(20): 12021–12041. <https://doi.org/10.1007/s00500-024-09948-w>
- [32] Chen, Y.Y., Zhang, L.B., Hu, J.Q., Liu, Z.Y. (2021). Optimization of distribution of emergency resources for emergency rescue points of oil and gas pipelines. In *E3S Web of Conferences*, 266: 01016. <https://doi.org/10.1051/e3sconf/202126601016>
- [33] Wang, L., Li, C., Dai, W., Zou, J., Xiong, H. (2021). QoE-driven and tile-based adaptive streaming for point

clouds. In ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Toronto, ON, Canada, pp. 1930-1934. <https://doi.org/10.1109/ICASSP39728.2021.9414121>

[34] Fleszar, K. (2022). A branch-and-bound algorithm for the quadratic multiple knapsack problem. *European Journal of Operational Research*, 298(1): 89-98. <https://doi.org/10.1016/j.ejor.2021.06.018>